Ατομική Διπλωματική Εργασία

ΠΡΟΣΟΜΟΙΩΣΗ ΚΑΙ ΑΞΙΟΛΟΓΗΣΗ ΠΑΡΑΛΛΗΛΩΝ ΑΛΓΟΡΙΘΜΩΝ ΣΤΟ ΠΕΡΙΒΑΛΛΟΝ XMT

Ελένη Παντελή

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2010

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΠΡΟΣΟΜΟΙΩΣΗ ΚΑΙ ΑΞΙΟΛΟΓΗΣΗ ΠΑΡΑΛΛΗΛΩΝ ΑΛΓΟΡΙΘΜΩΝ ΣΤΟ ΠΕΡΙΒΑΛΛΟΝ XMT

Ελένη Παντελή

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 2010

Ευχαριστίες

Σε αυτό το σημείο θα ήθελα να ευχαριστήσω τον υπεύθυνο ακαδημαϊκό που είχε την ευθύνη για την επίβλεψη αυτής της εργασίας, τον κύριο Χρύση Γεωργίου, για την καθοδήγηση και την βοήθεια που μου πρόσφερε. Οι συμβουλές που πήρα ήταν πολύ χρήσιμες και καθοριστικές για την ολοκλήρωση της εργασίας αυτής.

Περίληψη

Το γενικό θέμα αυτής της διπλωματικής εργασίας είναι ο παραλληλισμός και οι παράλληλοι αλγόριθμοι. Διανύουμε μια εποχή όπου ο παραλληλισμός βρίσκεται σε σπουδαία ακμή και αποδεικνύεται ότι έχει σημαντικά πλεονεκτήματα σε σχέση με τον σειριακό υπολογισμό. Αυτός είναι και ο λόγος που αποφασίστηκε το θέμα της διπλωματικής εργασίας να είναι η μελέτη, η υλοποίηση και η πειραματική αξιολόγηση κάποιων κύριων παράλληλων αλγορίθμων.

Συγκεκριμένα, υλοποιήθηκαν παράλληλοι αλγόριθμοι στη γλώσσα προγραμματισμού XMTC, στο μοντέλο παράλληλου υπολογισμού PRAM, και προσομοιώθηκαν με τον προσομοιωτή XMT. Έπειτα, πάρθηκαν μετρήσεις από την προσομοίωση των αλγορίθμων για την πειραματική τους αξιολόγηση. Βάσει των μετρήσεων αυτών εξάχθηκαν και καταγράφηκαν τα συμπεράσματα, που αφορούν τη σχέση μεταξύ θεωρητικής και πειραματική πολυπλοκότητας των αλγορίθμων, καθώς και την διαφορά απόδοσης διαφορετικών αλγορίθμων που επιλύουν το ίδιο παράλληλο πρόβλημα.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Σκοπός Διπλωματικής Εργασίας

Η διπλωματική εργασία αυτή έχει ως σκοπό την υλοποίηση και προσομοίωση παράλληλων αλγορίθμων. Η υλοποίηση των αλγορίθμων έγινε στη γλώσσα παράλληλου προγραμματισμού XMTC και η προσομοίωση με την χρήση του προσομοιωτή XMT (eXplicit Multi-Threading). Μετά από την προσομοίωση των αλγορίθμων, ακολούθησε η πειραματική τους αξιολόγηση με σκοπό την εξαγωγή συμπερασμάτων που αφορούν την απόδοση τους. Συγκεκριμένα, σκοπός της πειραματικής αξιολόγησης είναι η σύγκριση μεταξύ θεωρητικής και πρακτικής αξιολόγησης, καθώς και η σύγκριση μεταξύ βέλτιστων και μη βέλτιστων αλγορίθμων που επιλύουν το ίδιο πρόβλημα. Έτσι, θα διαπιστωθεί εάν όντως τα αποτελέσματα της πρακτικής αξιολόγησης επαληθεύουν την θεωρητική αξιολόγηση. Επιπλέον, θα καταλήξουμε στο συμπέρασμα εάν όντως οι βέλτιστοι αλγόριθμοι είναι καλύτεροι από τους μη βέλτιστους αλγόριθμους που επιλύουν το ίδιο πρόβλημα. Τέλος, επειδή η γλώσσα XMTC και ο προσομοιωτής XMT είναι νέα εργαλεία στοχεύουμε στην διερεύνηση των δυνατοτήτων τους, καθώς και των περιορισμών τους.

1.2 Μεθοδολογία Υλοποίησης

Η μεθοδολογία που ακολουθήθηκε για την υλοποίηση της διπλωματικής εργασίας αυτής είναι η ακόλουθη: Αρχικά, έγινε μελέτη του προσομοιωτή XMT, και η εκμάθηση της γλώσσας προγραμματισμού XMTC. Έπειτα, έγινε η εγκατάσταση του προσομοιωτή

XMT μαζί με την γλώσσα προγραμματισμού XMTC. Ακολούθησε η υλοποίηση και προσομοίωση κάποιων απλών αλγορίθμων στην γλώσσα XMTC για την καλύτερη της κατανόηση.

Στη συνέχεια έγινε η μελέτη και η υλοποίηση κάποιων επιλεγμένων παράλληλων αλγορίθμων. Συγκεκριμένα, υλοποιήθηκαν ο μη βέλτιστος και ο βέλτιστος αλγόριθμος παράλληλης άθροισης, ο αλγόριθμος παράλληλης μετάδοσης ενός στοιχείου, δύο μη βέλτιστοι αλγόριθμοι προθεματικού αθροίσματος, ο βέλτιστος αλγόριθμος προθεματικού αθροίσματος, δύο αλγόριθμοι παράλληλης αναζήτηση και ο βέλτιστος αλγόριθμος παράλληλης αναζήτησης.

Τέλος, έγινε η πειραματική αξιολόγηση των αλγορίθμων και η εξαγωγή συμπερασμάτων βάσει της αξιολόγησης αυτής.

1.3 Δομή Διπλωματικής Εργασίας

Το έγγραφο αυτό περιλαμβάνει οχτώ κεφάλαια. Το Κεφάλαιο 2 που ακολουθεί περιέχει μια εισαγωγή στον παραλληλισμό και στο μοντέλο PRAM για το οποίο σχεδιάστηκαν οι αλγόριθμοι που αναφέρθηκαν προηγουμένως. Έπειτα, στο Κεφάλαιο 3 γίνεται μια επεξήγηση του προσομοιωτή XMT και περιγράφονται τα κύρια χαρακτηριστικά της γλώσσας προγραμματισμού XMTC. Στα επόμενα κεφάλαια, δηλαδή στα Κεφάλαια 4 μέχρι 7, περιγράφονται τα βασικά στοιχεία των αλγόριθμων που υλοποιήθηκαν και αναλύεται η πολυπλοκότητα τους. Επιπλέον, γίνεται η πειραματική αξιολόγηση των αλγορίθμων και η εξαγωγή συμπερασμάτων. Στο τελευταίο κεφάλαιο, δηλαδή το Κεφάλαιο 8, αναφέρονται τα συμπεράσματα τα οποία εξάγονται από την δουλειά που έγινε για την ανάπτυξη και ολοκλήρωση της εργασίας αυτής.

Κεφάλαιο 2

Παραλληλισμός

2.1 Παράλληλος Υπολογισμός

Για πολλά χρόνια η επιστήμη της πληροφορικής μελετούσε την λειτουργία και τη χρήση σειριακών υπολογιστών. Επικεντρωνόταν στην επίλυση προβλημάτων με την χρήση σειριακών αλγορίθμων σε σειριακούς υπολογιστές. Στο σειριακό υπολογισμό υπάρχει ένας επεξεργαστής ο οποίος είναι υπεύθυνος για την σειριακή εκτέλεση όλων των εντολών ενός συγκεκριμένου προγράμματος.

Τα τελευταία χρόνια όμως αναπτύσσεται σημαντικά ο παραλληλισμός. Ο παραλληλισμός ασχολείται με την εκτέλεση προγραμμάτων σε υπολογιστές με περισσότερους από ένα επεξεργαστές, οι οποίοι είναι υπεύθυνοι για την ταυτόχρονη εκτέλεση διαφόρων εντολών των προγραμμάτων. Οι επεξεργαστές οι οποίοι εκτελούν τον ίδιο παράλληλο υπολογισμό πρέπει να βρίσκονται σε κοντινή απόσταση, είναι του ίδιου τύπου και φυσικά επιλύουν μαζί ένα κοινό πρόβλημα-στόχο. Όλοι οι επεξεργαστές που χρησιμοποιούνται για κάποιο υπολογισμό έχουν συλλογικά τεράστια υπολογιστική δύναμη.

Στόχος του παράλληλου υπολογισμού είναι η επίλυση δύσκολων υπολογιστικών προβλημάτων γρήγορα, σε χρόνο μικρότερο από αυτόν που χρειάζεται ο σειριακός υπολογισμός για ένα κοινό πρόβλημα.

Γενικά, ο παραλληλισμός είναι πολύ σημαντικός, αφού έχει γίνει απαραίτητος για τους υπολογισμούς και κυρίως για την ταχύτητα των υπολογισμών. Κάποια προβλήματα απαιτούν δύσκολους και πολύπλοκους υπολογισμούς τους οποίους ο σειριακός υπολογιστής δεν μπορεί να διεκπεραιώσει με επιτυχία ή αποδοτικά το πρόβλημα. Επιπλέον, υπάρχουν διάφοροι φυσικοί περιορισμοί που αφορούν τα κυκλώματα VLSI, οι οποίοι δεν βοηθούν την ταχύτητα εκτέλεσης σε σειριακό υπολογισμό. Αντίθετα, ο παράλληλος υπολογισμός είναι σε θέση να παρακάμπτει τους περιορισμούς αυτούς. Σήμερα, τα παράλληλα συστήματα έχουν χαμηλή τιμή, άρα συμφέρουν και οικονομικά [4, 5, 8].

Στα παράλληλα συστήματα χρησιμοποιούνται αλγόριθμοι που είναι σχεδιασμένοι για να εκτελούνται σε αυτά τα συστήματα και ονομάζονται *Παράλληλοι Αλγόριθμοι*. Για να καθορίσουμε το πόσο αποδοτικός είναι ο κάθε παράλληλος αλγόριθμος λαμβάνουμε υπόψη την πολυπλοκότητα του αλγορίθμου[4, 5, 8].

Για να υπολογίσουμε την πολυπλοκότητα ενός αλγορίθμου που επιλύει ένα πρόβλημα μεγέθους n, λαμβάνουμε υπόψη μας τις μετρικές που περιγράφονται πιο κάτω [4, 5] :

- Πλήθος Επεξεργαστών – $P(n)$

  Είναι ο αριθμός των επεξεργαστών που χρησιμοποιούνται για την επίλυση ενός προβλήματος μεγέθους n με παράλληλο αλγόριθμο.
- Παράλληλος Χρόνος Εκτέλεσης – $T_p(n)$

  Είναι ο χρόνος που χρειάζεται για την εκτέλεση ενός αλγορίθμου, από ένα αριθμό p επεξεργαστών που εργάζονται παράλληλα, για την επίλυση ενός προβλήματος μεγέθους n.
- Κόστος – $C_p(n)$

  Είναι το γινόμενο του πλήθους των επεξεργαστών που χρειάστηκαν για την επίλυση ενός προβλήματος μεγέθους n και του παράλληλου χρόνου εκτέλεσης.

  $C_p(n) = P(n) * T_p(n)$

Όπως αναφέρθηκε προηγουμένως, στόχος του παράλληλου υπολογισμού είναι η γρήγορη επίλυση ενός δεδομένου προβλήματος. Για αυτό επιθυμούμαι την σχεδίαση βέλτιστων παράλληλων αλγορίθμων, δηλαδή αλγόριθμους που εκτελούνται με τον μικρότερο δυνατό χρόνο για ένα συγκεκριμένο πρόβλημα. Για να είναι κάποιος

παράλληλος αλγόριθμος *βέλτιστος*, πρέπει το κόστος εκτέλεσης του να είναι της τάξης του χρόνου εκτέλεσης του πιο γρήγορου σειριακού αλγόριθμου που επιλύει το ίδιο πρόβλημα. Συγκεκριμένα, ένας αλγόριθμος είναι βέλτιστος όταν $C_p(n) = O(T^*(n))$, όπου $T^*(n)$ είναι ο χρόνος εκτέλεσης του γρηγορότερου σειριακού αλγόριθμου.

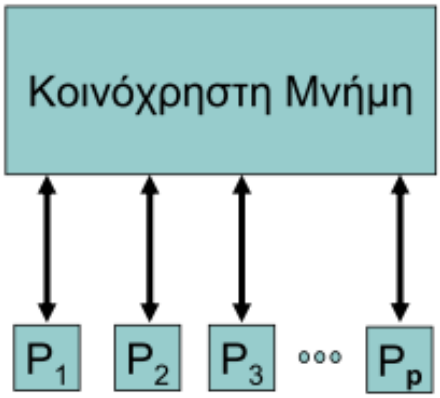
2.2 Μοντέλο PRAM

Το PRAM (**P**arallel **R**andom **A**ccess **M**achine) είναι μια μηχανή τυχαίας προσπέλασης που χρησιμοποιείται στο παράλληλο υπολογισμό. Είναι ένα μοντέλο κοινόχρηστης μνήμης αποτελούμενο από ένα μέγιστο αριθμό επεξεργαστών, όπου ο καθένας έχει τη δική του τοπική μνήμη.

Αποτελεί κατηγορία του μοντέλου παράλληλου υπολογισμού SIMD (**S**ingle **I**nstruction Stream, **M**ultiple **D**ata Stream), στο οποίο όλοι οι επεξεργαστές εκτελούν την ίδια εντολή αλλά με διαφορετικά δεδομένα εισόδου. Ο κάθε επεξεργαστής έχει τη δική του τοπική μνήμη όπου μπορεί να αποθηκεύσει τιμές στις οποίες έχει πρόσβαση μόνο ο ίδιος. Όλοι οι επεξεργαστές επικοινωνούν μεταξύ τους μέσω μιας κοινόχρηστης μνήμης ή ενός δικτύου αλληλοσύνδεσης.

Στο μοντέλο PRAM, το οποίο είναι το μοντέλο το οποίο θα χρησιμοποιηθεί για την υλοποίηση των αλγορίθμων που θα περιγραφούν στα επόμενα κεφάλαια, χρησιμοποιείται μόνο η κοινόχρηστη μνήμη. Η κοινόχρηστη μνήμη χρησιμεύει για την έμμεση επικοινωνία μεταξύ των επεξεργαστών, αφού εδώ αποθηκεύονται μεταβλητές οι οποίες είναι ορατές από όλους τους διαθέσιμους επεξεργαστές. Συγκεκριμένα, στη κοινόχρηστη μνήμη αποθηκεύονται τα δεδομένα εισόδου του προβλήματος το οποίο καλούμε να λύσουμε, καθώς και τα αποτελέσματά εξόδου. Επιπλέον, οι επεξεργαστές μπορούν να αποθηκεύσουν οποιαδήποτε άλλη ενδιάμεση τιμή η οποία επιθυμούμε να είναι ορατή και στους υπόλοιπους επεξεργαστές. Οι επεξεργαστές έχουν ομοιόμορφη πρόσβαση στη κοινόχρηστη μνήμη σε σταθερό χρόνο, δηλαδή μπορούν να διαβάσουν ή να γράψουν μια τιμή σε χρόνο Θ(1). Στο μοντέλο PRAM όλοι οι επεξεργαστές είναι πανομοιότυποι και λειτουργούν συγχρονισμένα. Δηλαδή, όταν εκτελείται μια εντολή από κάποιο επεξεργαστή, τότε η ίδια εντολή εκτελείται από όλους τους υπόλοιπους αλγορίθμους την ίδια χρονική στιγμή. Γενικά, οι επεξεργαστές εκτελούν ταυτόχρονα

όλα τα βήματα του αλγορίθμου. Είναι όμως δυνατό ένα σύνολο επεξεργαστών να μην εκτελεί κάποια εντολή για ένα χρονικό διάστημα επειδή δεν χρειάζεται για την σωστή επίλυση του προβλήματος. Σε αυτή την περίπτωση θεωρούμε ότι οι επεξεργαστές εκτελούν την εντολή no-op (no operation) [4, 5].



Σχήμα 2.1 – Μοντέλο PRAM [5]

Ο κάθε επεξεργαστής μπορεί να γράψει ένα μήνυμα σε μια κυψελίδα της κοινόχρηστης μνήμης και οποιοσδήποτε άλλος μπορεί να το διαβάσει και να το επεξεργαστεί. Μπορεί να υπάρξει πάντα ταυτόχρονη πρόσβαση στη κοινόχρηστη μνήμη από περισσότερους από ένα επεξεργαστές, με τη προϋπόθεση ότι δεν έχουν πρόσβαση στην ίδια κυψελίδα μνήμης.

Υπάρχουν διάφοροι τύποι PRAM που αφορούν τους περιορισμούς ταυτόχρονης πρόσβασης στη κοινόχρηστη μνήμη. Οι τύποι αυτοί είναι οι ακόλουθοι [4, 5] :

- **EREW**: **E**xclusive **R**ead, **E**xclusive **W**rite
  Σε αυτή την περίπτωση δεν επιτρέπεται ταυτόχρονη ανάγνωση ή γραφή της ίδιας κυψελίδας της κοινόχρηστης μνήμης από περισσότερους από ένα επεξεργαστές. Το μοντέλο αυτό του PRAM είναι το πιο αδύνατο, αφού περιορίζει τις λειτουργίες των επεξεργαστών.
- **CREW**: **C**oncurrent **R**ead, **E**xclusive **W**rite
  Μπορούν όλοι οι επεξεργαστές να διαβάσουν ταυτόχρονα τιμή στην ίδια κυψελίδας μνήμης, αλλά μόνο ένας επεξεργαστής μπορεί να γράψει σε μια κυψελίδα μνήμης ανά χρονική στιγμή.

- **ERCW**: **E**xclusive **R**ead, **E**xclusive **W**rite

  Αυτός ο τύπος επιτρέπει την ταυτόχρονη γραφή στην ίδια κυψελίδα μνήμης από όλους τους επεξεργαστές, αλλά δεν επιτρέπει το ταυτόχρονο διάβασμα της ίδιας κυψελίδας από περισσότερους από ένα επεξεργαστές. Το μοντέλο αυτό υπάρχει μόνο σε θεωρητικό επίπεδο.
- **CRCW**: **C**oncurrent **R**ead, **E**xclusive **W**rite

  Στη συγκεκριμένη περίπτωση δεν υπάρχουν καθόλου περιορισμοί που να αφορούν την ταυτόχρονη πρόσβαση των επεξεργαστών στην ίδια κυψελίδα της κοινόχρηστης μνήμης. Όταν υπάρχει ταυτόχρονη γραφή στην ίδια κυψελίδα μνήμης από διαφορετικούς επεξεργαστές τότε υπάρχει η πιθανότητα να δημιουργηθούν προβλήματα. Για αυτό υπάρχουν υπο-μοντέλα του CRCW PRAM τα οποία χειρίζονται την ταυτόχρονη γραφή στην ίδια κυψελίδα μνήμης. Τα μοντέλα αυτά είναι τα ακόλουθα:

  - Common CRCW PRAM

    Σε αυτό το μοντέλο πρέπει όλοι οι επεξεργαστές που γράφουν την ίδια στιγμή, στην ίδια κυψελίδα της κοινόχρηστης μνήμης να γράψουν την ίδια τιμή.
  - Arbitrary CRCW PRAM

    Σε αυτό το μοντέλο επιλέγεται τυχαία ένας από τους επεξεργαστές που επιθυμεί να γραφή στην κοινόχρηστη μνήμη και αποθηκεύει την δική του τιμή, ενώ οι τιμές των άλλων επεξεργαστών ακυρώνονται.
  - Priority CRCW PRAM

    Σε αυτό το μοντέλο επιλέγεται ένας από τους επεξεργαστές που επιθυμεί γραφή στην κοινόχρηστη μνήμη, αυτός με την μεγαλύτερη προτεραιότητα. Η προτεραιότητα συνήθως καθορίζεται από το αναγνωριστικό των επεξεργαστών, δηλαδή το μοναδικό προσωπικό τους αριθμό. Για παράδειγμα, η προτεραιότητα μπορεί να καθορίζεται βάσει του αναγνωριστικού σε *αύξουσα* σειρά, δηλαδή ο επεξεργαστής με το μεγαλύτερο αναγνωριστικό να έχει την μεγαλύτερη προτεραιότητα. Για παράδειγμα, στην περίπτωση που χρησιμοποιούνται n επεξεργαστές, η προτεραιότητα είναι η ακόλουθη: $P_n, \dots, P_3, P_2, P_1$.

Στην περίπτωση όπου υπάρχει ταυτόχρονη ανάγνωση από περισσότερους από ένα επεξεργαστές, της ίδιας κυψελίδας μνήμης, τότε δεν παρουσιάζεται πρόβλημα. Αυτό οφείλεται στο ότι ο κάθε επεξεργαστής απλά "βλέπει" την τιμή της κοινόχρηστης κυψελίδας και την αποθηκεύει στην τοπική του μνήμη.

Τα διάφορα μοντέλα του PRAM που περιγράφηκαν προηγουμένως έχουν διαφορετική δυναμικότητα και περιορισμούς. Το EREW είναι το πιο περιοριστικό μοντέλο, ενώ το Priority CRCW είναι το πιο ελαστικό. Η ιερατικότητα της δυναμικότητας των μοντέλων του PRAM αναφέρεται πιο κάτω:

1. EREW
2. CREW
3. Common CRCW
4. Arbitrary CRCW
5. Priority CRCW

Όταν ένας αλγόριθμος σχεδιαστεί σε ένα πιο περιοριστικό μοντέλο, τότε μπορεί να υλοποιηθεί σε οποιοδήποτε πιο ελαστικό μοντέλο με την ίδια πολυπλοκότητα χρόνου και κόστους. Σε αντίθετη περίπτωση, δηλαδή όταν ένας αλγόριθμος είναι σχεδιασμένος σε πιο ελαστικό μοντέλο τότε δύναται να υλοποιηθεί σε πιο περιοριστικό μοντέλο με κάποιο επιπλέον χρόνο εκτέλεσης ή κόστος.

Εξαιτίας των περιορισμών τους, το κάθε μοντέλο PRAM έχει διαφορετικές απαιτήσεις που αφορούν την υλοποίηση ενός αλγορίθμου. Για αυτό το λόγο είναι απαραίτητο τα άτομα που σχεδιάζουν ένα αλγόριθμο στο μοντέλο PRAM να έχει υπόψη του τον τύπο του μοντέλου στο οποίο θα υλοποιηθεί ο αλγόριθμος, καθώς και τους περιορισμούς του μοντέλου αυτού.

2.3 Σημαντικότητα Μοντέλου PRAM

Το μοντέλο PRAM είναι ένα πολύ σημαντικό μοντέλο παράλληλου υπολογισμού με αρκετά πλεονεκτήματα. Είναι ένα καλό αλγοριθμικό μοντέλο που προσφέρεται για καλύτερη και γενική μελέτη των παράλληλων αλγορίθμων. Το μοντέλο αυτό παρέχει

συγχρονισμό των επεξεργαστών και η επικοινωνία μεταξύ των επεξεργαστών είναι πολύ απλή. Έτσι, επικεντρώνεται στον τρόπο υλοποίησης των παράλληλων αλγορίθμων και στα χαρακτηριστικά του προβλήματος που καλείται να επιλύσει ένας αλγόριθμος, αγνοώντας λεπτομέρειες που αφορούν τον συγχρονισμό και την επικοινωνία μεταξύ των επεξεργαστών. Επιπλέον, αλγόριθμοι σχεδιασμένοι στο μοντέλο PRAM μπορούν εύκολα να προσαρμοστούν και να εκτελεστούν σε άλλα μοντέλα παράλληλου υπολογισμού. Γενικά, είναι ένα σαφές μοντέλο παράλληλου υπολογισμού όπου οι αναθέσεις των εργασιών σε επεξεργαστές και οι λειτουργίες που εκτελούν είναι ξεκάθαρες.

Κεφάλαιο 3

Προσομοιωτής και Γλώσσα Προγραμματισμού

3.1 Προσομοιωτής XMT

Το XMT (E**X**plicit **M**ulti-**T**hreading) [2] είναι ένα υπολογιστικό περιβάλλον το οποίο ξεκίνησε να αναπτύσσεται το 1997, στο Πανεπιστήμιο του Maryland. Το XMT αναπτύχθηκε για παράλληλες εφαρμογές στο μοντέλο PRAM, και συγκεκριμένα στο Arbitrary CRCW PRAM. Εξομοιώνει ακριβώς το μοντέλο PRAM όπως είναι στην πραγματικότητα με την κοινόχρηστη μνήμη και ένα σύνολο επεξεργαστών.

Το XMT βασίζεται σε μια αρχιτεκτονική ενός PRAM-On-Chip υπολογιστή που μπορεί να υποστηρίξει πέραν των 1000 επεξεργαστών στο chip. Το PRAM-On-Chip αποτελεί

την πρώτη προσπάθεια για πρακτική υλοποίηση του μοντέλου PRAM με κοινόχρηστη μνήμη. Η ανάγκη για αποδοτική εκτέλεση των παράλληλων αλγορίθμων, οδήγησε στην ανάπτυξη ενός πρωτοτύπου της αρχιτεκτονικής αυτής για το XMT, μια μηχανή με 64 επεξεργαστές που βασίζεται στην τεχνολογία FPGA. Η μηχανή αυτή, την οποία βλέπουμε στο Σχήμα 3.1, αποτελείται από τρία *FPGA chips* και περιλαμβάνει ένα Master Thread Control Unit (MTCU) που χειρίζεται κάποιες ειδικές εντολές, παράλληλα TCU σε *clusters*, οχτώ *on-chip cache modules*, ένα *memory controller* (MC), ένα *global register file* (GRF) και ένα *prefix sum unit*. [2, 6, 7].

Σχήμα 3.1 – Μηχανή 64 επεξεργαστών, πρωτοτύπου FPGA

Το XMT παρέχει τη γλώσσα προγραμματισμού XMTC για την ανάπτυξη προγραμμάτων στο εργαλείο αυτό. Η γλώσσα XMTC είναι μια προέκταση της γλώσσας C, με κάποιες επιπλέον εντολές που αφορούν τον παραλληλισμό, αλλά και με κάποιους περιορισμούς σε σχέση με την C [1, 2, 3].

Τα προγράμματα που αναπτύσσονται στη γλώσσα XMTC μπορούν να εκτελεστούν είτε απευθείας στη μηχανή που είναι βασισμένη στο πρωτότυπο FPGA, είτε σε ένα "cycle accurate" προσομοιωτή (XMT Simulator) [1].

Όπως προαναφέρθηκε, τα παράλληλα προγράμματα μπορούν να εκτελεστούν σε μηχανή 64 επεξεργαστών, η οποία χρησιμοποιεί την τεχνολογία FPGA. Τα προγράμματα αυτά μπορούν να εκτελεστούν και σε άλλες αρχιτεκτονικές χρησιμοποιώντας ένα μεταγλωττιστή που μετατρέπει τα XMTC προγράμματα σε OpenMP [2, 6, 7].

Στη περίπτωση του XMT προσομοιωτή, μπορούν να προσομοιωθούν προγράμματα με τον ίδιο τρόπο που θα εκτελούταν σε μια πραγματική PRAM μηχανή. Συγκεκριμένα, ο προσομοιωτής αυτός χρησιμοποιείται για να προσομοιώνει τους αλγόριθμους όπως θα εκτελούνταν στη μηχανή που βασίζεται στο πρωτότυπο FPGA. Επιτρέπει την χρήση 1024 επεξεργαστών κατά την προσομοίωση ενός αλγορίθμου. Σε αυτή την περίπτωση πριν την προσομοίωση το πρόγραμμα πρέπει να μεταγλωττιστεί με τον μεταγλωττιστή της γλώσσας XMTC, που ονομάζεται *xmtcc* [2, 7].

3.2 Γλώσσα Προγραμματισμού XMTC

Η γλώσσα που θα χρησιμοποιηθεί για την υλοποίηση των παράλληλων αλγορίθμων είναι η XMTC. Η XMTC, όπως αναφέρθηκε και προηγουμένως είναι μια γλώσσα προγραμματισμού που χρησιμοποιείται από το εργαλείο XMT και επιτρέπει παράλληλο προγραμματισμό στο μοντέλο PRAM. Όπως σε κάθε PRAM μοντέλο, υπάρχει μια κοινόχρηστη μνήμη μέσω της οποίας επικοινωνούν όλοι οι επεξεργαστές ενός προγράμματος και ο κάθε επεξεργαστής έχει τη δική του τοπική μνήμη.

Στη γλώσσα XMTC επιτρέπεται η ταυτόχρονη ανάγνωση και γραφή στην ίδια κυψελίδα κοινόχρηστης μνήμης από περισσότερους από ένα επεξεργαστές, αλλά είναι στην ευθύνη του προγραμματιστή να χειριστεί την κατάσταση αυτή. Συγκεκριμένα, χρησιμοποιείται το μοντέλο Arbitrary CRCW PRAM. Στην περίπτωση ταυτόχρονης γραφής το τελικό περιεχόμενο της κοινόχρηστης μνήμης εξαρτάται από τον επεξεργαστή που θα γράψει την τιμή. Άρα, η ταυτόχρονη γραφή επηρεάζει το τελικό αποτέλεσμα που θα πάρουμε από την κοινόχρηστη μνήμη. Ο προγραμματιστής πρέπει να έχει υπόψη του το γεγονός αυτό, έτσι ώστε να σχεδιάζει αλγόριθμους που να αντιμετωπίζουν το πρόβλημα που παρουσιάζεται. Η ταυτόχρονη ανάγνωση δεν επηρεάζει το περιεχόμενο της κοινόχρηστης μνήμης, αλλά είναι δυνατό να προκαλέσει χαμηλότερη απόδοση. Η γλώσσα XMTC παρέχει εντολές και τεχνικές για των χειρισμό καταστάσεων ταυτόχρονης πρόσβασης στη μνήμη.

Η γλώσσα αυτή είναι παρόμοια με την γλώσσα σειριακού προγραμματισμού C. Στην XMTC χωρίζεται ο σειριακός από τον παράλληλο υπολογισμό με την χρήση

συγκεκριμένων εντολών. Κάποιος μπορεί να εκτελέσει ένα σειριακό πρόγραμμα υλοποιημένο στη γλώσσα C και στη γλώσσα XMTC.

Η XMTC διατηρεί τα κύρια χαρακτηριστικά και τη δομή της γλώσσας C, αλλά υπάρχουν κάποια επιπλέον χαρακτηριστικά και εντολές έτσι ώστε να γίνει κατάλληλη γλώσσα για ανάπτυξη παράλληλων αλγορίθμων. Υπάρχουν όμως και κάποια σημαντικά χαρακτηριστικά της γλώσσας C που δεν μπορούν να χρησιμοποιηθούν ή δεν λειτουργούν πάντα ορθά στην XMTC. Τα χαρακτηριστικά αυτά θα αναφερθούν σε επόμενο υποκεφάλαιο.

Τέλος, αξίζει να σημειωθεί ότι η γλώσσα XMTC δεν βρίσκεται ακόμα στο τελικό στάδιο της και για αυτό δεν είναι εγγυημένο ότι κάποια χαρακτηριστικά της γλώσσας μπορούν να λειτουργήσουν ορθά. Υπάρχουν όμως κάποιες λειτουργίες και περιορισμοί της γλώσσας που βρίσκονται σε στάδιο ανάπτυξης από τους ερευνητές που ασχολούνται με το θέμα αυτό [1, 2, 3].

3.3 Παράδειγμα στη γλώσσα XMTC

Σε αυτό το σημείο θα παρουσιαστεί ο κώδικας για ένα απλό παράλληλο αλγόριθμο που υλοποιήθηκε στη γλώσσα XMTC, και συγκεκριμένα για τον αλγόριθμο παράλληλης μετάδοσης στο μοντέλο EREW PRAM.

Ο αλγόριθμος αυτός λαμβάνει ως δεδομένο εισόδου το πλήθος των θέσεων της κοινόχρηστης μνήμης n, στις οποίες θα μεταδοθεί το στοιχείο της πρώτης θέσης ενός πίνακα A της κοινόχρηστης μνήμης. Σε αυτό τον αλγόριθμο χρησιμοποιούνται $n/2$ επεξεργαστές για την μετάδοση της τιμής A[1]. Αρχικά, ο πρώτος επεξεργαστής διαβάζει την A[1] και την αντιγράφει στην δεύτερη κυψελίδα, έπειτα δύο επεξεργαστές διαβάζουν την A[1] και την αντιγράφουν στις δύο επόμενες κενές θέσεις. Η διαδικασία επαναλαμβάνεται μέχρι να μεταδοθεί το A[1] και στις n θέσεις της κοινόχρηστης μνήμης. Σε κάθε βήμα διπλασιάζεται ο αριθμός των επεξεργαστών που χρησιμοποιούνται, το ίδιο και οι θέσεις στις οποίες μεταδίδεται η τιμή A[1]. Περισσότερες λεπτομέρειες για τον αλγόριθμο αυτό θα αναφερθούν στο Κεφάλαιο 5.

Ο κώδικας για τον αλγόριθμο της παράλληλης μετάδοσης στο μοντέλο PRAM είναι ο ακόλουθος:

```
#include <xmtc.h>
#include "broadcast.h"


int main()
{
        A[1] = 1;

        //Parallel Mode
       spawn(1, n/2)
       {
            //Local Variable
            int k;
             k = 1;

            while(k < n)
            {
                 if($ <= k)
                      A[$ + k] = A[$];
                 k = k * 2;
            }
       } //end of parallel mode

       printf("%d elements have been broadcasted\n", n);
}
```

Στον πιο πάνω αλγόριθμο φαίνεται η δομή και τα κύρια χαρακτηριστικά της γλώσσας XMTC, τα οποία θα περιγραφούν στο επόμενο υποκεφάλαιο. Επιπλέον, είναι εμφανής η απλότητα της γλώσσας και η ομοιότητα της με την γλώσσα σειριακού προγραμματισμού C.

3.4 Χαρακτηριστικά της γλώσσας XMTC

Όπως παρατηρούμε και από τον πιο πάνω κώδικα, η δομή της γλώσσας XMTC είναι η ίδια με τη δομή της γλώσσας C. Δηλαδή, πριν την έναρξη του κώδικα πρέπει να γίνεται η ενσωμάτωση κάποιων απαραίτητων για την εκτέλεση του προγράμματος βιβλιοθηκών και στη συνέχεια ακολουθεί η συνάρτηση “main()” στην οποία γίνονται οι

βασικές λειτουργίες του προγράμματος. Τόσο εντός, όσο και εκτός της συνάρτησης "main()" μπορούν να δηλωθούν μεταβλητές οι οποίες αποτελούν τις κοινόχρηστες μεταβλητές του προγράμματος. Η δήλωση συναρτήσεων από τον προγραμματιστή είναι επιτρεπτή, αλλά η κλήση τους επιτρέπεται μόνο εκτός του παράλληλου κομματιού του κώδικα [2].

3.4.1 Βιβλιοθήκες

Αρχικά, είναι απαραίτητη η εισαγωγή κάποιων βιβλιοθηκών στα παράλληλα προγράμματα που θα εκτελεστούν στη γλώσσα XMTC. Η πιο σημαντική βιβλιοθήκη που πρέπει να ενσωματωθεί στο πρόγραμμα είναι η βιβλιοθήκη <*xmtc.h*>. Χωρίς τη βιβλιοθήκη αυτή η εκτέλεση του προγράμματος είναι αδύνατη, αφού επιτρέπει την χρήση όλων των βασικών εντολών της XMTC. Μια ακόμη σημαντική βιβλιοθήκη είναι αυτή που επιτρέπει την εκτύπωση μηνυμάτων σε κάποια κονσόλα είναι η <*xmtio.h*>, η οποία ενσωματώνεται αυτόματα από τον μεταγλωττιστή. Άρα, δεν πρέπει να εισάγεται στον κώδικα από τον προγραμματιστή. Σε κάθε πρόγραμμα είναι απαραίτητη η ενσωμάτωση βιβλιοθήκης που δημιουργείται από τον προγραμματιστή που περιέχει κάποιες κοινόχρηστες μεταβλητές του προγράμματος, καθώς και τις αρχικοποιήσεις τους.
Η ενσωμάτωση των βιβλιοθηκών στο πρόγραμμα μπορεί να γίνει είτε με την εντολή "#include" στον κώδικα, είτε με την εντολή "-include" κατά την μεταγλώττιση του προγράμματος [1, 2].

3.4.2 Μεταβλητές

Οι κοινόχρηστες μεταβλητές μπορούν να δηλωθούν στη συνάρτηση "main()", είτε έξω από την "main()" στην αρχή του προγράμματος. Επιπλέον, είναι δυνατό να δηλωθούν οι κοινόχρηστες μεταβλητές μέσω ενός εργαλείου που επιτρέπει την εισαγωγή εξωτερικών δεδομένων στο πρόγραμμα. Οι μεταβλητές αυτές εισάγονται στο πρόγραμμα μέσω βιβλιοθήκης.

Οι τοπικές μεταβλητές του κάθε επεξεργαστή δηλώνονται μόνο στο παράλληλο κομμάτι του προγράμματος. Για παράδειγμα, στον πιο πάνω κώδικα για την παράλληλη

μετάδοση, μια τέτοια μεταβλητή είναι η k, η οποία δηλώνεται στην αρχή του παράλληλου κομματιού. Για την μεταβλητή αυτή δημιουργείται ένα αντίγραφο για τον κάθε επεξεργαστή. Περισσότερες πληροφορίες θα αναφερθούν στη συνέχεια.

Οι δηλώσεις των μεταβλητών γίνονται όπως ακριβώς και στη γλώσσα προγραμματισμού C. Όμως στη γλώσσα XMTC υπάρχουν περιορισμοί που αφορούν τους τύπους δεδομένων. Επιτρέπεται μόνο η δήλωση ακεραίων μεταβλητών, δηλαδή τύπου "int" και "long". Επίσης, είναι δυνατή η δήλωση μονοδιάστατων ή δυσδιάστατων πινάκων, καθώς και "struct" των τύπων που αναφέρθηκαν προηγουμένως. Η χρήση δεικτών επιτρέπεται αλλά αποθαρρύνεται αφού είναι πιθανό να προκαλέσει διάφορα προβλήματα που αφορούν τον μεταγλωττιστή.

Εκτός από τα είδη των μεταβλητών που αναφέρθηκαν, υπάρχει και ένα είδος μεταβλητών που δεν υπάρχει στη γλώσσα C και χρησιμοποιείται μόνο για παράλληλο υπολογισμό. Το είδος μεταβλητών αυτό είναι το "psBaseReg" και χρησιμοποιείται στις εντολές ps, που θα επεξηγηθούν στη συνέχεια. Οι μεταβλητές του τύπου αυτού πρέπει να δηλώνονται ως global μεταβλητές, δηλαδή εκτός της "main()". Υπάρχει όμως ο περιορισμός ότι μόνο έξι μεταβλητές τύπου "psBaseReg" μπορούν να δηλωθούν. Είναι προσβάσιμες τόσο από το σειριακό, όσο και από το παράλληλο μέρος ενός προγράμματος. Στο παράλληλο μέρος όμως είναι προσβάσιμο μόνο από μια συγκεκριμένη εντολή, την "ps" [1, 2].

3.4.3 Εντολή spawn

Η εντολή "spawn(start_thread, end_thread)" καθορίζει το παράλληλο μέρος του κώδικα. Το παράλληλο μέρος ξεκινά με την εμφάνιση της εντολής αυτής και τελειώνει στο τέλος του spawn block. Όλες οι εντολές που βρίσκονται στο κομμάτι αυτό εκτελούνται παράλληλα. Η εντολή αυτή καθορίζει και τον αριθμό των επεξεργαστών που εκτελούν παράλληλα το περιεχόμενο του κομματιού αυτού. Η πρώτη παράμετρος της εντολής καθορίζει το αναγνωριστικό του πρώτου επεξεργαστή και η δεύτερη παράμετρος το αναγνωριστικό του τελευταίου. Για παράδειγμα, στον κώδικα του Υποκεφαλαίου 3.3 η εντολή "spawn(1, $n/2$)" καθορίζει ότι βρίσκονται σε λειτουργία οι επεξεργαστές με αναγνωριστικό από 1 μέχρι $n/2$. Το πλήθος των επεξεργαστών που

χρησιμοποιούνται ισούται με (end_thread - start_thread + 1), δηλαδή στην περίπτωση του παραδείγματος μας $n/2$.

Ο καθένας από τους επεξεργαστές εκτελεί ανεξάρτητα το ίδιο κομμάτι κώδικα που περιλαμβάνεται στο παράλληλο μέρος. Οι μεταβλητές που δηλώνονται στο μέρος αυτό είναι τοπικές μεταβλητές και ο κάθε επεξεργαστής έχει το δικό του αντίγραφο της μεταβλητής αυτής. Ο αριθμός όμως των τοπικών μεταβλητών που μπορούν να δηλωθούν δεν πρέπει να είναι μεγάλος. Αυτό οφείλεται στον μικρό αριθμό καταχωρητών του παράλληλου **T**hread **C**ontrol **U**nit (TCU) που κρατούν τις μεταβλητές αυτές. Η μόνη λύση μέχρι τώρα για αυτό το πρόβλημα είναι το σπάσιμο του κώδικα σε περισσότερα spawn blocks. Στο μέλλον αναμένεται να αυξηθεί ο αριθμός των TCU καταχωρητών, καθώς και να παρέχεται από τον μεταγλωττιστή το σπάσιμο του κώδικα σε περισσότερα spawn blocks [1, 2].

Υπάρχει επίσης περιορισμός που αφορά τις εντολές στο εσωτερικό του παράλληλου μέρους. Συγκεκριμένα, ο αριθμός των assembly εντολών στο κάθε spawn block που δημιουργούνται μετά την μεταγλώττιση του προγράμματος δεν πρέπει να ξεπερνά τις 1000. Στην περίπτωση που ο αριθμός των εντολών αυτών ξεπερνά τις 1000 είναι αδύνατη η εκτέλεση του προγράμματος, αν και στη μεταγλώττιση δεν παρουσιάζεται λάθος. Η μόνη λύση μέχρι τώρα για αυτό το πρόβλημα είναι το σπάσιμο του κώδικα σε περισσότερα spawn blocks.

Όπως προαναφέρθηκε, δεν επιτρέπεται η κλήση συναρτήσεων από το εσωτερικό του spawn. Επίσης, δεν επιτρέπεται ούτε η χρήση φωλιασμένων spawns, όμως επιτρέπεται και ενθαρρύνεται η χρήση περισσότερων από ένα ανεξάρτητων spawn bocks [1, 2].

3.4.4 Σύμβολο $

Σε κάθε παράλληλο μέρος του προγράμματος, σε κάθε επεξεργαστή δίνεται ένα μοναδικό αναγνωριστικό που καθορίζεται από τις τιμές που παίρνει ως παραμέτρους η εντολή spawn.

Το σύμβολο “$” δίνει στον προγραμματιστή πρόσβαση σε ένα επεξεργαστή και αντιπροσωπεύει το αναγνωριστικό του κάθε επεξεργαστή. Δηλαδή, κατά την εκτέλεση του παράλληλου μέρους στα σημεία όπου υπάρχει το σύμβολο “$” παίρνουν μια τιμή από πεδίο των παραμέτρων της εντολής “spawn”. Για παράδειγμα, στον κώδικα του Υποκεφαλαίου 3.3, η πράξη “$A[\$ + k] = A[\$];$” έχει ως αποτέλεσμα την πρόσβαση στις θέσεις μνήμης που αντιστοιχούν στο αναγνωριστικό του καθένα από τους $n/2$ επεξεργαστές που είναι σε λειτουργία.

Η τιμή του συμβόλου αυτού δεν μπορεί να αλλάξει και γενικά δεν μπορεί να του ανατεθεί μια οποιαδήποτε τιμή από τον χρήστη [1, 2].

3.4.5 Συνάρτηση printf

Αν και η χρήση συναρτήσεων που δηλώνονται από τον προγραμματιστή επιτρέπεται στο σειριακό κομμάτι, είναι αδύνατη η κλήση συναρτήσεων του συστήματος.
Μόνη εξαίρεση είναι η συνάρτηση “printf()” η οποία μπορεί να χρησιμοποιηθεί για εκτύπωση αποτελεσμάτων σε κάποια κονσόλα, μόνο όμως στο σειριακό κομμάτι του κώδικα όπως συμβαίνει και με τις υπόλοιπες συναρτήσεις. Η χρήση της συνάρτηση αυτή είναι παρόμοια με την συνάρτηση “printf()” της γλώσσας C. Υπάρχει όμως περιορισμός στον αριθμό των παραμέτρων που μπορεί να έχει η κάθε printf συνάρτηση στο προσομοιωτή. Συγκεκριμένα, επιτρέπεται να υπάρχουν μόνο τρεις “%d” εκφράσεις. Αφού ο μόνος τύπος δεδομένων που επιτρέπεται είναι οι integer, μόνο οι εκφράσεις “%d” μπορούν να χρησιμοποιηθούν. Αν και δεν υποστηρίζονται οι τύποι char, είναι δυνατή η εκτύπωση σταθερών συμβολοσειρών. Στη περίπτωση της μηχανής FPGA δεν υπάρχει οποιοσδήποτε περιορισμός που αφορά των αριθμό των παραμέτρων, αλλά υπάρχει περιορισμός στον αριθμό των χαρακτήρων που τυπώνονται [2].

3.4.6 Εντολή ps

Η γλώσσα XMTC παρέχει μια εντολή η οποία προβαίνει σε υπολογισμό προθεματικής άθροισης. Η εντολή αυτή έχει την ακόλουθη δομή:

ps(int local_integer, psBaseReg ps_base)

Για την εκτέλεση της εντολής αυτής απαιτείται η χρήση δύο μεταβλητών, μιας τύπου “int” και μιας τύπου “psBaseReg”. Η μεταβλητή τύπου “int” πρέπει να δηλώνεται μόνο μέσα σε κάποιο παράλληλο κομμάτι κώδικα, δηλαδή πρέπει να είναι τοπική μεταβλητή. Πρέπει επίσης να αρχικοποιείται μόνο με τιμή 0 ή 1. Η μεταβλητή τύπου πρέπει να είναι καθολική μεταβλητή.

Κάθε φορά που εκτελείται η συγκεκριμένη εντολή προστίθεται η τιμή της πρώτης μεταβλητής στη τιμή της δεύτερης και η δεύτερη μεταβλητή κρατά το αποτέλεσμα. Η τιμή που είχε η δεύτερη μεταβλητή πριν την πρόσθεση, αποθηκεύεται στην πρώτη μεταβλητή. Αυτή η διαδικασία γίνεται αυτόματα κάθε φορά που εμφανίζεται η εντολή ps. Η εντολή αυτή είναι δυνατό να εμφανιστεί μόνο σε κάποιο παράλληλο κομμάτι του κώδικα και συνήθως χρησιμοποιείται για τον χειρισμό περιπτώσεων ταυτόχρονης γραφής στη κοινόχρηστη μνήμη [2].

3.4.7 Εργαλείο Μνήμης για δεδομένα εισόδου

Ο μόνος τρόπος για εισαγωγή δεδομένων σε ένα XMTC πρόγραμμα είναι μέσω ενός εργαλείου, του *MemMapCreator*. Το εργαλείο αυτό επιτρέπει την στατική δέσμευση μνήμης για μεταβλητές ενός προγράμματος. Επιπλέον, είναι δυνατή η αρχικοποίηση των μεταβλητών αυτών. Αποτέλεσμα της χρήσης αυτού του εργαλείου είναι η δημιουργία κάποιων αρχείων με τις πληροφορίες που αφορούν τα δεδομένα εισόδου, τα οποία πρέπει να ενσωματωθούν στον υπόλοιπο κώδικα. Οι μεταβλητές που υπάρχουν στο αρχείο, καθώς και οι τιμές τους μπορούν να χρησιμοποιηθούν κανονικά στο πρόγραμμα.

Το είδος των αρχείων που δημιουργούνται μετά την χρήση του MemMapCreator είναι τα ακόλουθα:

- Header File (.h)

  Το αρχείο αυτό περιλαμβάνει τις δηλώσεις των κοινόχρηστων μεταβλητών για τις οποίες έγινε η δέσμευση μνήμης. Στην περίπτωση που κάποια από τις μεταβλητές είναι πίνακας, υπάρχει και η δήλωση μιας μεταβλητής που αντιπροσωπεύει το μέγεθος του πίνακα. Το αρχείο αυτό λειτουργεί ως

βιβλιοθήκη για αυτό μπορεί να ενσωματωθεί στο κώδικα με τον ίδιο τρόπο που ενσωματώνονται οι βιβλιοθήκες.

- Binary File (.xbo)

  Στο αρχείο αυτό υπάρχουν οι αρχικοποιήσεις των μεταβλητών που είναι δηλωμένες στο header file. Μια μεταβλητή αρχικοποιείται με μια τιμή που δίνει ο προγραμματιστής μέσω του MemMapCreator. Όλα τα στοιχεία ενός πίνακα μπορούν να αρχικοποιηθούν με την τιμή 0, με τυχαίες τιμές από το 0 μέχρι μια τιμή που επιλέγει ο προγραμματιστής, καθώς και μέσω ενός αρχείου στο οποίο ο προγραμματιστής δίνει τις τιμές που επιθυμεί.

  Το αρχείο αυτό, δίνεται ως είσοδος στο μεταγλωττιστή κατά τη διάρκεια της μεταγλώττισης του προγράμματος.

- Text File (.txt)

  Το αρχείο αυτό δείχνει τα περιεχόμενα του binary file, αλλά δεν χρησιμοποιείται από τον προσομοιωτή ή από τη μηχανή FPGA. Μπορεί να χρησιμοποιηθεί μόνο για σκοπούς ελέγχου από τον προγραμματιστή [2].

Παράδειγμα της χρήσης του εργαλείου μνήμης αυτού είναι ο κώδικας στο Υποκεφάλαιο 3.3. Η δέσμευση μνήμης για το πλήθος των δεδομένων εισόδου και του πίνακα με τις θέσεις στις οποίες μεταδίδεται ο αριθμός, δεν γίνεται μέσα στον κώδικά αλλά με την βοήθεια αυτού του εργαλείου. Επιπλέον, το εργαλείο αυτό χρησιμοποιείται για την αρχικοποίηση των πιο πάνω μεταβλητών. Για να είναι ορατές οι μεταβλητές αυτές κατά την μεταγλώττιση του αλγορίθμου γίνεται στον κώδικα η εισαγωγή του header file “broadcast.h”.

Στη συνέχεια ακολουθεί το περιεχόμενο του “broadcast.h”, έτσι όπως δημιουργείται μετά από την χρήση του εργαλείου μνήμης.

```
/**************************************
 Automatically Generated Header File

      DO NOT MODIFY

 Contains following Variables :

Name      : n
Type      : int
Dimension : 1
Size [0]  : 1
Source    : 1024

Name      : A
Type      : int
Dimension : 1
Size [0]  : 1024
Source    : 0

**************************************/
extern int n;

extern int A[1024];
extern int A_dim0_size;
```

Στο πιο πάνω αρχείο, περιέχονται οι δηλώσεις των κοινόχρηστων μεταβλητών για τις οποίες έγινε η δέσμευση μνήμης, καθώς και του μεγέθους του πίνακα (A_dim0_size).

3.4.8 XMTC Serializer

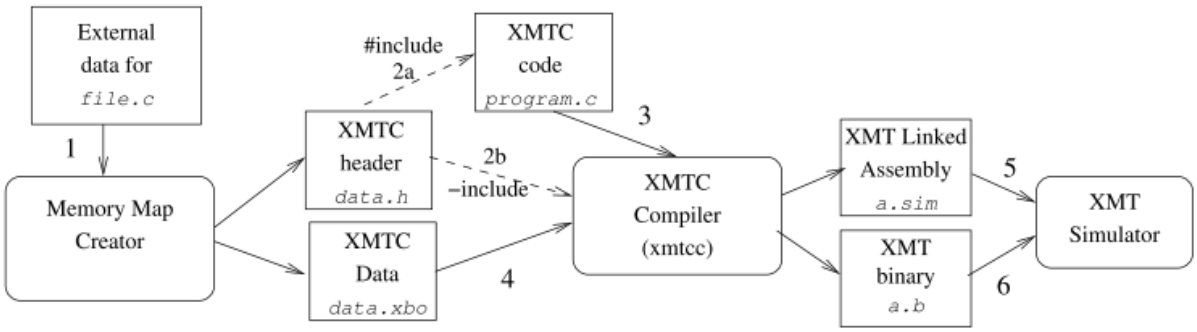
Επειδή ακόμη ο μεταγλωττιστής της XMTC δεν είναι στο τελικό του στάδιο είναι πιθανό να εμφανίζει λάθη στο πρόγραμμα που προέρχονται από τον ίδιον ίδιο τον μεταγλωττιστή και δεν είναι λάθη του προγραμματιστή. Για να μπορέσουμε να ελέξουμε την κατάσταση αυτή, δηλαδή να κατανοήσουμε αν ένα λάθος το οποίο παρουσιάστηκε οφείλεται στο προγραμματιστή ή όχι μπορούμε να χρησιμοποιήσουμε τον XMTC Serializer.

Ο XMTC Serializer μετατρέπει τον παράλληλο κώδικα σε σειριακό, ο οποίος μπορεί να μεταγλωττιστεί με ένα από τους γνωστούς μεταγλωττιστές της C, όπως τον gcc. Έτσι, μπορεί να γίνει έλεγχος στα ουσιαστικά λάθη του προγραμματιστή [2].

3.5 Μεταγλώττιση και Προσομοίωση Προγράμματος

Για να εκτελεστεί ένα XMTC πρόγραμμα πρέπει πρώτα να μεταγλωττιστεί με την χρήση του μεταγλωττιστή της XMTC, *xmtcc*. Έπειτα, πρέπει να προσομοιωθεί με την χρήση του *xmtsim*, που είναι ο προσομοιωτής του XMT.

Στο ακόλουθο σχήμα φαίνεται η διαδικασία που πρέπει να ακολουθεί για την μεταγλώττιση και την προσομοίωση ενός XMTC προγράμματος.



Σχήμα 3.2 – Διαδικασία μεταγλώττισης και προσομοίωσης XMTC προγράμματος [2]

Αφού ο προγραμματιστής ολοκληρώσει την εγγραφή του κώδικα, πρέπει να καθορίσει τα δεδομένα εισόδου. Αυτό γίνεται με την χρήση του εργαλείου Memory Map Creator, το οποίο δίνει την δυνατότητα δημιουργίας και αρχικοποίησης κοινόχρηστων μεταβλητών.

Το εργαλείο αυτό όπως αναφέρθηκε προηγουμένως, παράγει τρία αρχεία τα δύο από τα οποία πρέπει να χρησιμοποιηθούν για την ορθή εκτέλεση του προγράμματος. Το header file πρέπει να εισαχθεί στον κώδικα στην αρχή του αρχείου, στο σημείο όπου δηλώνονται οι βιβλιοθήκες με την χρήση της εντολής "#include". Μία εναλλακτική λύση είναι η ενσωμάτωση του αρχείου κατά την μεταγλώττιση με την εντολή "-include". Το δεύτερο αρχείο που παράγεται από το Memory Map Creator, δηλαδή το binary file, πρέπει να δοθεί ως είσοδος στον μεταγλωττιστή με την εντολή "-binload".

Χρησιμοποιώντας την εντολή xmtcc μπορούμε να μεταγλωττίσουμε το πρόγραμμα, αφού εφαρμόσουμε όσα αναφέρθηκαν προηγουμένως. Ένα παράδειγμα, της εντολής μεταγλώττισης χρησιμοποιώντας τις ονομασίες που δίνονται στο πιο πάνω σχήμα είναι το ακόλουθο:

```
> xmtcc program.c –o a data.xbo
```

Από την μεταγλώττιση παράγονται δύο αρχεία τα a.sim και a.b. Το a.b είναι το εκτελέσιμο αρχείο, ενώ το a.sim είναι το assembly αρχείο που δημιουργείται. Τα αρχεία αυτό χρησιμοποιούνται κατά την προσομοίωση, με την πιο κάτω εντολή:

```
> xmtsim a.sim –binload a.b -cycle
```

Η εντολή "-cycle" που χρησιμοποιείται κατά την προσομοίωση είναι υπεύθυνη για την ταυτόχρονη εκτέλεση των εντολών του προγράμματος. Κατά την προσομοίωση ενός αλγορίθμου με την εντολή αυτή παρέχεται και ο *χρόνος εκτέλεσης* του αλγορίθμου. Ο χρόνος εκτέλεσης που δίνει η προσομοίωση με την εντολή "-cycle", είναι κοντά με τον χρόνο που θα έδιναν οι φυσικοί XMT επεξεργαστές [2].

3.6 Πλατφόρμα Υλοποίησης και Αξιολόγησης

Η υλοποίηση και αξιολόγηση των παράλληλων αλγορίθμων έγινε μέσω ενός Virtual Machine με το λειτουργικό σύστημα "Ubuntu", του *Sun xVM VirtualBox*. Το Virtual Machine αυτό είναι εγκατεστημένο σε ένα φορητό υπολογιστή με το λειτουργικό σύστημα "Windows Vista Business", με μνήμη RAM 1.00GB, με επεξεργαστή "Intel Core Duo" και συχνότητα 2.20GHz.

Κεφάλαιο 4

Αλγόριθμος Παράλληλης Άθροισης

4.1 Πρόβλημα

Το πρόβλημα της άθροισης περιλαμβάνει μια λίστα από *n* αριθμούς $x_1, x_2, \ldots, x_n$ οι οποίοι αποτελούν τα δεδομένα του προβλήματος. Πρέπει με κάποιο αποτελεσματικό τρόπο να υπολογίσουμε το άθροισμα S των αριθμών αυτών.

4.2 Σειριακός Αλγόριθμος

Το πρόβλημα της άθροισης λύνεται εύκολα σειριακά, με την εκτέλεση $n-1$ επαναλήψεων και τη χρήση ενός επεξεργαστή.

Σε κάθε επανάληψη, ο επεξεργαστής διαβάζει ένα αριθμό από την λίστα και έπειτα τον προσθέτει στο μέχρι εκείνη τη στιγμή άθροισμα. Μέχρι την τελευταία επανάληψη έχουν προστεθεί όλοι οι αριθμοί και έχουμε το τελικό αποτέλεσμα.

Ο χρόνος εκτέλεσης του σειριακού αλγόριθμου είναι $\Theta(n)$. Όπως παρατηρούμε στο πιο κάτω κομμάτι κώδικα αρχικά διαβάζουμε το πρώτο στοιχείο της λίστας για να προσθέσουμε σε αυτό τα υπόλοιπα στοιχεία. Το βήμα αυτό απαιτεί σταθερό χρόνο $\Theta(1)$. Στη συνέχεια, με τη χρήση βρόγχου διαβάζεται κάθε φορά ένας αριθμός και

προστίθεται στα υπόλοιπα στοιχεία, σε σταθερό χρόνο Θ(1). Το περιεχόμενο του βρόγχου εκτελείται $n-1$ φορές έτσι ώστε να διαβαστούν τα $n-1$ στοιχεία που δεν είχαν διαβαστεί στο πρώτο βήμα. Έτσι, η πολυπλοκότητα εκτέλεσης του σειριακού αλγορίθμου άθροισης είναι Θ(n).

Ο σειριακός αλγόριθμος για την παράλληλη άθροιση είναι ο ακόλουθος [5]:

set S = x_1
for i = 2 to n do
    S = S + x_i
end do

Ο χρόνος εκτέλεσης στην περίπτωση που μόλις περιγράφτηκε είναι Θ(n), αφού έχουμε στη διάθεση μας μόνο ένα επεξεργαστή για την εκτέλεση των απαραίτητων πράξεων. Στη περίπτωση όμως που μπορούμε να έχουμε περισσότερους από ένα επεξεργαστές, ο χρόνος εκτέλεσης θεωρητικά είναι μικρότερος αφού υπάρχει η δυνατότητα παράλληλης εκτέλεσης των πράξεων.

4.3 Παράλληλος Αλγόριθμος

Στην περίπτωση της επίλυσης του προβλήματος της άθροισης με παράλληλο αλγόριθμο χρησιμοποιούνται περισσότεροι από ένας επεξεργαστής και συγκεκριμένα $n/2$ επεξεργαστές, οπού n είναι το σύνολο των στοιχείων της λίστας. Τα στοιχεία της λίστας βρίσκονται αποθηκευμένα σε $n-1$ συνεχόμενες κυψελίδες της κοινόχρηστης μνήμης. Έτσι, επεξεργαζόμαστε τη λίστα ως πίνακα με στοιχεία στις θέσεις 1 μέχρι n.

Στη πρώτη εκτέλεση του βρόγχου του αλγορίθμου, ο κάθε ένας από τους $n/2$ επεξεργαστές διαβάζει και αθροίζει δύο διαφορετικούς αριθμούς από τον πίνακα με τα στοιχεία στη κοινόχρηστη μνήμη. Έπειτα, τα αθροίσματα αποθηκεύονται στη παρόν θέση του πίνακα της κοινόχρηστης μνήμης, αντικαθιστώντας τις αρχικές τιμές της λίστας.

Σε κάθε επόμενο βήμα, ο αριθμός των επεξεργαστών που είναι σε λειτουργία μειώνονται στους μισούς. Οι επεξεργαστές αυτοί αθροίζουν τους αριθμούς που

αποθηκεύτηκαν στον πίνακα της κοινόχρηστης μνήμης στο προηγούμενο βήμα. Η διαδικασία αυτή επαναλαμβάνεται μέχρι να απομείνουν δύο αριθμοί, τους οποίους αθροίζει ένας επεξεργαστής και δίνει το τελικό αποτέλεσμα.

Το μοντέλο το οποίο χρησιμοποιείται για την ανάπτυξη του αλγορίθμου είναι το EREW, αφού δεν υπάρχει ταυτόχρονη ανάγνωση ή γραφή στην κοινόχρηστη μνήμη. Η ανάγνωση στοιχείων από την κοινόχρηστη μνήμη γίνεται βάση του μοναδικού αναγνωριστικού i του κάθε επεξεργαστή. Το ίδιο ισχύει και για την αποθήκευση του αθροίσματος στην κοινόχρηστη μνήμη. Έτσι, αφού δεν υπάρχει ανάγνωση ή γραφή στην κοινόχρηστη μνήμη την ίδια χρονική στιγμή από διαφορετικούς επεξεργαστές, τότε χρησιμοποιείται το μοντέλο EREW.

Ο παράλληλος αλγόριθμος άθροισης είναι ο πιο κάτω [5] :

```
Processors i = 1 to n/2 do in parallel
        shared array S[1..n]
        private j, a, b
        set j = 1
        while (i <= 2^j)
                a = S[2*i-1]
                b = S[2*i]
                S[i] = a + b
                j = j + 1
        end while
end do
```

Ο πιο πάνω αλγόριθμος εκτελείται σε χρόνο $\Theta(\log n)$. Ο χρόνος αυτός οφείλεται στις $\Theta(\log n)$ επαναλήψεις του βρόγχου που γίνονται παράλληλα το πολύ για n/2 επεξεργαστές. Οι επεξεργαστές που βρίσκονται σε λειτουργία κάθε χρονική στιγμή καθορίζονται από την συνθήκη ($i <= 2^j$) όπου κάθε φορά λειτουργούν επεξεργαστές ανάλογα με το αναγνωριστικό τους. Όπως παρατηρούμε, μετά από κάθε επανάληψη, μειώνονται στους μισούς οι επεξεργαστές που βρίσκονται σε λειτουργία και άρα, χρειάζεται χρόνος $\Theta(\log n)$ για την εκτέλεση των επαναλήψεων.

Οι πράξεις στην κάθε επανάληψη απαιτούν σταθερό χρόνο Θ(1), άρα συνολικά χρειάζεται χρόνος Θ($\log n$) για την εκτέλεση του αλγορίθμου παράλληλης άθροισης.
Όπως αναφέρθηκε προηγουμένως, για τον αλγόριθμο αυτό χρειάζονται n/2 επεξεργαστές και αφού ο χρόνος εκτέλεσης του είναι Θ($\log n$), τότε το κόστος εκτέλεσης του αλγορίθμου είναι $(n/2)$ *Θ($\log n$). Άρα, το συνολικό κόστος εκτέλεσης του αλγόριθμου παράλληλης άθροισης είναι Θ($n\log n$).

Ο αλγόριθμος αυτός δεν είναι βέλτιστος, αφού το κόστος εκτέλεσης του είναι μεγαλύτερο από τον χρόνο εκτέλεσης που απαιτείται κατά την σειριακή εκτέλεση του αλγορίθμου. Στόχος μας είναι η υλοποίηση ενός αλγορίθμου παράλληλης άθροισης, ο οποίος να είναι βέλτιστος.

4.4 Βέλτιστος Αλγόριθμος

Στη περίπτωση του βέλτιστου αλγόριθμου πρέπει να χρησιμοποιηθούν λιγότεροι επεξεργαστές ώστε να μειωθεί το κόστος. Έτσι, για την επίλυση του προβλήματος άθροισης με βέλτιστο τρόπο χρησιμοποιούνται $n/\log n$ επεξεργαστές.

Ο αλγόριθμος χωρίζεται σε δύο βήματα. Αρχικά, ο κάθε ένας από τους $n/\log n$ επεξεργαστές εκτελεί σειριακό άθροισμα σε ένα σύνολο στοιχείων της κοινόχρηστης μνήμης. Έπειτα, $(n/\log n)/2$ επεξεργαστές εκτελούν τον αλγόριθμο παράλληλης άθροισης που περιγράφτηκε προηγουμένως.

Στο πρώτο βήμα οι $n/\log n$ επεξεργαστές λειτουργούν παράλληλα και ο καθένας εκτελεί σειριακό άθροισμα για $\log n$ διαφορετικά στοιχεία από την κοινόχρηστη μνήμη. Συγκεκριμένα, ο πρώτος επεξεργαστής αθροίζει το πρώτο σύνολο από $\log n$ στοιχεία της κοινόχρηστης μνήμης, ο δεύτερος το πρώτο σύνολο από $\log n$ στοιχεία, μέχρι τον $n/\log n$ επεξεργαστή που αθροίζει το τελευταίο $(n/\log n)$ σύνολο από $\log n$ στοιχεία. Μόλις ολοκληρωθεί η σειριακή άθροιση, ο κάθε επεξεργαστής αποθηκεύει το άθροισμα που υπολόγισε στη θέση της κοινόχρηστης μνήμης που του αντιστοιχεί, δηλαδή στη θέση με αριθμό το αναγνωριστικό του επεξεργαστή. Έτσι, απομένουν $n/\log n$ στοιχεία για τα οποία πρέπει να υπολογιστεί το άθροισμα.

Στο δεύτερο βήμα, χρησιμοποιούνται $(n/\log n)/2$ επεξεργαστές για την εκτέλεση του παράλληλου αλγορίθμου αφού το σύνολο των στοιχείων που πρέπει να αθροιστούν είναι $n/\log n$. Η διαδικασία άθροισης είναι ακριβώς η ίδια με αυτήν που περιγράφτηκε στο προηγούμενο μέρος, αλλά με διαφορετικό αριθμό των επεξεργαστών και στοιχείων.

Το μοντέλο το οποίο χρησιμοποιείται για την ανάπτυξη του αλγορίθμου είναι το EREW, αφού δεν υπάρχει ταυτόχρονη ανάγνωση ή γραφή στην κοινόχρηστη μνήμη. Η ανάγνωση στοιχείων από την κοινόχρηστη μνήμη γίνεται βάση του μοναδικού αναγνωριστικού i του κάθε επεξεργαστή, τόσο στο πρώτο όσο και στο δεύτερο βήμα του αλγορίθμου. Το ίδιο ισχύει και για την αποθήκευση του αθροίσματος στην κοινόχρηστη μνήμη. Έτσι, αφού δεν υπάρχει ανάγνωση ή γραφή στην κοινόχρηστη μνήμη την ίδια χρονική στιγμή από διαφορετικούς επεξεργαστές, τότε χρησιμοποιείται το μοντέλο EREW.

Ο χρόνος εκτέλεσης του βέλτιστου αλγορίθμου είναι ο ίδιος με τον χρόνο εκτέλεσης του αλγορίθμου παράλληλης άθροισης, δηλαδή $\Theta(\log n)$. Αυτό οφείλεται στο πρώτο βήμα του βέλτιστου αλγορίθμου, όπου ο κάθε επεξεργαστής αθροίζει σειριακά $\log n$ στοιχεία. Αφού όπως και αναφέρθηκε στην αρχή της ενότητας αυτής, ο χρόνος εκτέλεσης του σειριακού αλγόριθμου για n στοιχεία είναι $\Theta(n)$, τότε για $\log n$ στοιχεία χρειάζεται χρόνος $\Theta(\log n)$.

Ο χρόνος εκτέλεσης για το δεύτερο βήμα είναι επίσης $\Theta(\log n)$. Το δεύτερο βήμα χρησιμοποιεί τον αλγόριθμο παράλληλης άθροισης που περιγράφηκε προηγουμένως και εκτελείται σε χρόνο $\Theta(\log n)$ για n στοιχεία. Άρα, ο αλγόριθμος αυτός που αθροίζει στην περίπτωσή μας $n/\log n$ στοιχεία, έχει χρόνο εκτέλεσης $\Theta(\log(n/\log n))$. Με βάση τις ιδιότητες των λογαρίθμων ο χρόνος $\Theta(\log(n/\log n))$ ισούται με $\Theta(\log n - \log(\log n))$. Επομένως, ο χρόνος εκτέλεσης για το δεύτερο βήμα του αλγορίθμου είναι της τάξης $\Theta(\log n)$. Άρα, αφού και τα δύο μέρη του αλγορίθμου έχουν χρόνο εκτέλεσης $\Theta(\log n)$, συνολικά χρειάζεται χρόνος $\Theta(\log n)$ για την εκτέλεση του βέλτιστου αλγορίθμου παράλληλης άθροισης.

Όπως αναφέρθηκε προηγουμένως, αυτός ο αλγόριθμος χρησιμοποιεί το πολύ $n/\log n$ επεξεργαστές. Αφού ο χρόνος εκτέλεσης είναι $\Theta(\log n)$, τότε το κόστος εκτέλεσης είναι

$(n/\log n)$ * $\Theta(\log n)$. Άρα, το συνολικό κόστος εκτέλεσης του αλγορίθμου αυτού είναι $\Theta(n)$.

Δηλαδή, ο βέλτιστος αλγόριθμος παράλληλης άθροισης υπολογίζει το άθροισμα n στοιχείων με κόστος $\Theta(n)$ ο οποίος είναι της τάξης του χρόνου στον οποίο εκτελείται ο βέλτιστος σειριακός αλγόριθμος άθροισης. Έτσι, θεωρητικά επιτύχαμε την υλοποίηση ενός αλγόριθμου που υπολογίζει παράλληλα την άθροιση n στοιχείων με βέλτιστο κόστος.

4.5 Πειραματική Αξιολόγηση Αλγορίθμων

Οι δύο αλγόριθμοι παράλληλης άθροισης που μελετήσαμε υλοποιήθηκαν στο μοντέλο EREW PRAM, αναπτύχθηκαν με την χρήση της γλώσσας XMTC και προσομοιώθηκαν με την χρήση του XMT για να πάρουμε τα αποτελέσματα του παράλληλου υπολογισμού. Ακολουθούν οι μετρήσεις που πάρθηκαν από την προσομοίωση των δύο αλγορίθμων και τα συμπεράσματα στα οποία καταλήγουμε βάσει των μετρήσεων. Οι μετρήσεις πάρθηκαν από την εκτέλεση του κάθε αλγορίθμου με διαφορετικό αριθμό δεδομένων εισόδου n κάθε φορά. Τα δεδομένα του προβλήματος εισάχθηκαν στο υπόλοιπο πρόγραμμα μέσω αρχείων που δημιουργήθηκαν με τη βοήθεια του εργαλείου μνήμης που παρέχει η XMTC. Οι τιμές των δεδομένων εισόδου καθορίζονται τυχαία από το εργαλείο μνήμης και είναι ακέραιοι αριθμοί από το 0 μέχρι το 1000.

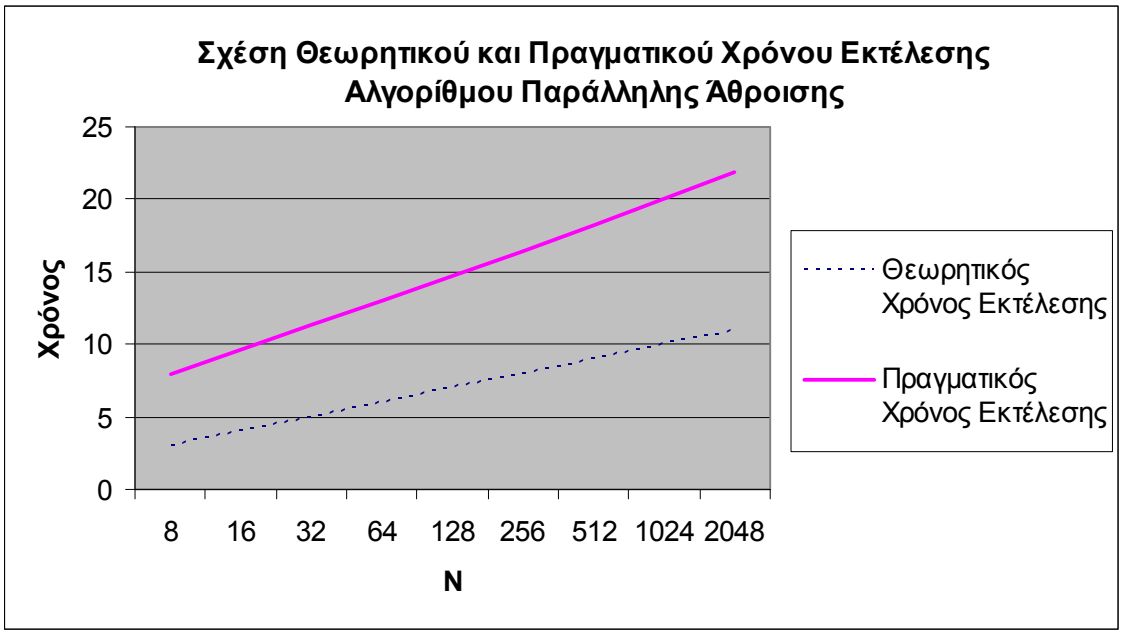
Αλγόριθμος Παράλληλης Άθροισης

Στον ακόλουθο πίνακα μπορούμε να δούμε τον χρόνο και το κόστος εκτέλεσης του μη βέλτιστου αλγορίθμου παράλληλης άθροισης με n δεδομένα εισόδου και με τη χρήση P επεξεργαστών. Ο χρόνος εκτέλεσης δίνεται από τον προσομοιωτή, μετά το τέλος της προσομοίωσης του αλγορίθμου. Ο χρόνος εκτέλεσης στον πιο κάτω πίνακα παρουσιάζεται σε milliseconds (msec). Το κόστος δεν παρέχεται από τον προσομοιωτή, αλλά υπολογίζεται βάσει του γινομένου του χρόνου εκτέλεσης που δίνει ο προσομοιωτής και του αριθμού των επεξεργαστών που χρησιμοποιούνται σε κάθε περίπτωση. Επίσης, στον πίνακα φαίνεται ο χρόνος $(\log n)$ που θεωρητικά πρέπει να έχει ο αλγόριθμος με βάση την ασυμπτωτικής ανάλυσης.

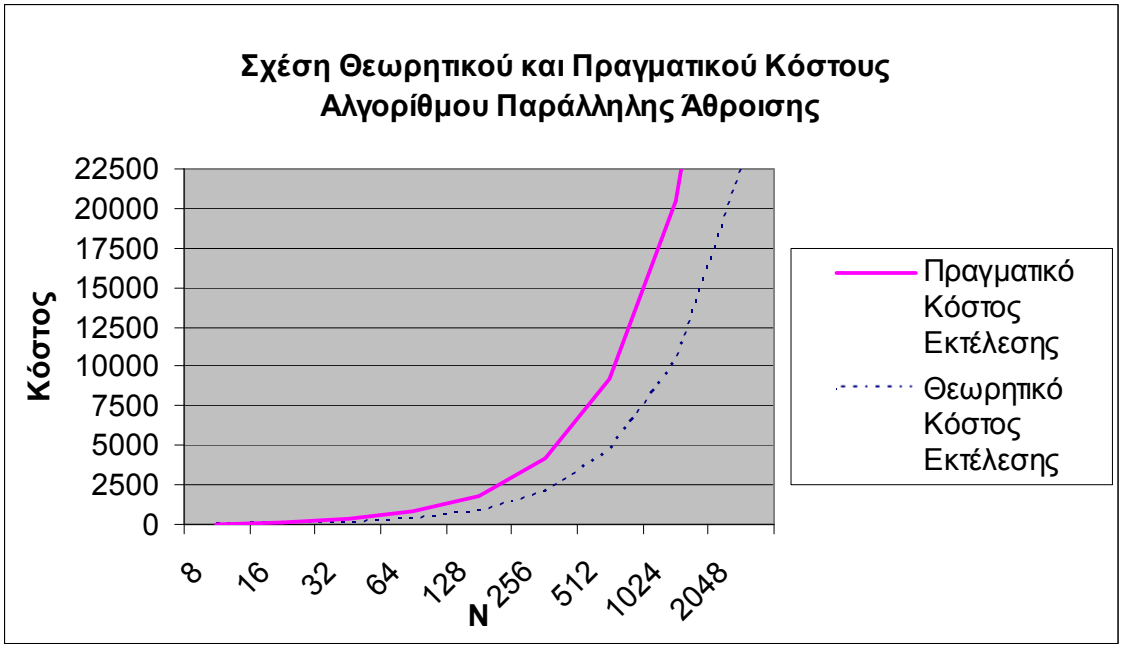
n	P	$T = \log n$	Χρόνος	Κόστος
8	4	3	7.92	31.68
16	8	4	9.63	77.04
32	16	5	11.34	181.44
64	32	6	13.05	417.6
128	64	7	14.76	944.64
256	128	8	16.47	2108.16
512	256	9	18.18	4654.08
1024	512	10	19.99	10234.88
2048	1024	11	21.89	22415.36

Πίνακας 4.1 – Μετρήσεις Προσομοίωσης

Ακολουθούν δύο γραφικές παραστάσεις, η Γραφική 4.1 στην οποία αναπαρίσταται ο χρόνος εκτέλεσης της θεωρητικής ανάλυσης μαζί με τον χρόνο εκτέλεσης που πάρθηκε από τις μετρήσεις που φαίνονται στον πιο πάνω πίνακα, και η Γραφική 4.2 στην οποία αναπαρίσταται το θεωρητικό και πραγματικό κόστος.



Γραφική 4.1 – Σχέση Θεωρητικού και Πραγματικού Χρόνου Εκτέλεσης



Γραφική 4.2 – Σχέση Θεωρητικού και Πραγματικού Κόστους Εκτέλεσης

Από τον πιο πάνω Πίνακα 4.1 παρατηρούμε ότι καθώς αυξάνεται ο αριθμός των δεδομένων εισόδου, αυξάνεται και ο χρόνος εκτέλεσης του αλγορίθμου. Συγκεκριμένα, από την Γραφική 4.1 παρατηρούμε ότι η αύξηση του πραγματικού χρόνου εκτέλεσης είναι λογαριθμικής τάξης, όπως συμβαίνει και στην θεωρητική ανάλυση. Από την Γραφική 4.2 στην οποία αναπαρίσταται η τάση αύξησης του κόστους σε σχέση με την αύξηση των δεδομένων, παρατηρούμε ότι η αύξηση του κόστους σε σχέση με τα δεδομένα εισόδου είναι της τάξης $n \log n$, όπως είναι και το θεωρητικό κόστος. Άρα, τόσο ο χρόνος εκτέλεσης του αλγορίθμου, όσο και το κόστος εκτέλεσης του, εξαρτώνται από το πλήθος των δεδομένων εισόδου. Αυτό επαληθεύεται στο γεγονός ότι ο χρόνος και το κόστος δηλώνονται ασυμπτωτικά με βάση τα δεδομένα εισόδου.

Σε αυτό το σημείο ας παρατηρήσουμε τη διαφορά που παρουσιάζεται μεταξύ του θεωρητικού χρόνου εκτέλεσης και του πραγματικού χρόνου εκτέλεσης του αλγορίθμου. Όπως βλέπουμε, ο χρόνος που απαιτείται για την ολοκλήρωση της πραγματικής εκτέλεσης του προγράμματος είναι μεγαλύτερος από τον θεωρητικό χρόνο. Συγκεκριμένα, ο πραγματικός χρόνος εκτέλεσης κυμαίνεται περίπου σε διπλάσιες τιμές από τις τιμές του θεωρητικού χρόνου. Το γεγονός αυτό είναι λογικό γιατί ο θεωρητικός χρόνος είναι ασυμπτωτικός, δηλαδή δεν λαμβάνει υπόψη του ένα σύνολο από σταθερές και πράξεις. Αν και αυτές οι σταθερές δεν επηρεάζουν τον ασυμπτωτικό χρόνο

εκτέλεσης, παίζουν ρόλο στον καθορισμό του πραγματικού χρόνου εκτέλεσης. Το σημαντικό όμως, όπως αναφέρθηκε και προηγουμένως είναι ότι ο πραγματικός χρόνος που λήφθηκε από τις προσομοιώσεις είναι της ίδιας μορφής με τον θεωρητικό χρόνο.

Παρατηρούμε επιπλέον ότι και το κόστος αυξάνεται κατά $n \log n$ όπως απαιτεί και η θεωρητική ανάλυση, όταν αυξάνονται τα δεδομένα και ο χρόνος. Η αύξηση είναι λογική αφού το κόστος υπολογίζεται από τον πραγματικό χρόνο εκτέλεσης του αλγορίθμου. Αφού το κόστος εξαρτάται από τον χρόνο εκτέλεσης του αλγορίθμου, όσα παρατηρήθηκαν στην προηγούμενη παράγραφο για τη σχέση μεταξύ θεωρητικού και πραγματικού χρόνου εκτέλεσης του αλγορίθμου ισχύουν και για το κόστος.

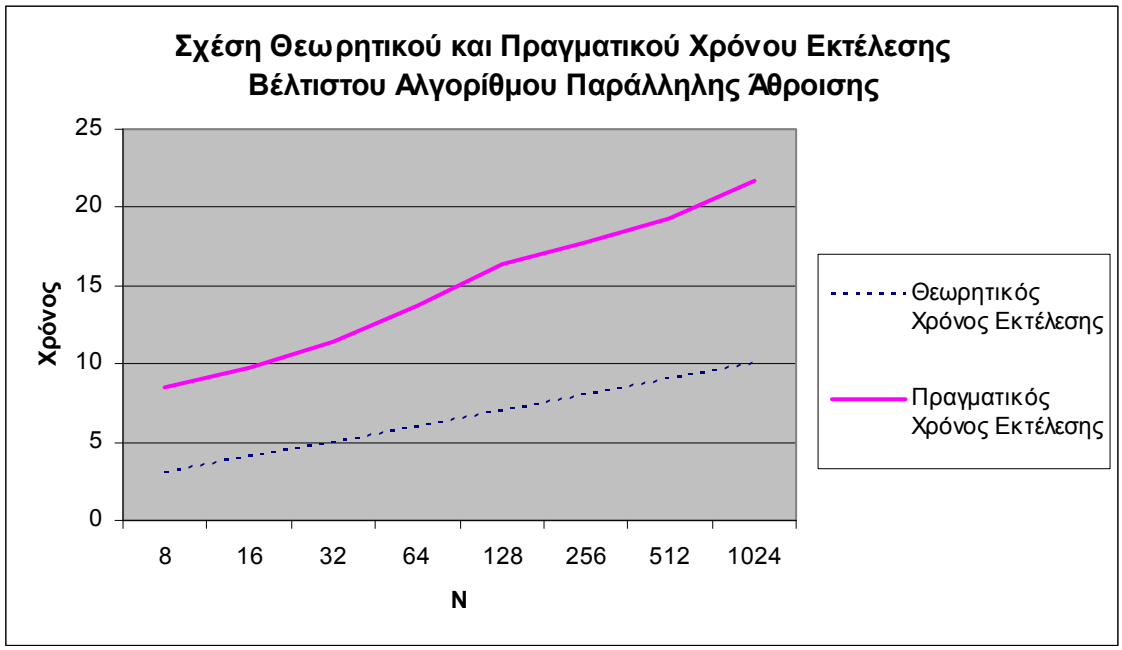
Βέλτιστος Αλγόριθμος Παράλληλης Άθροισης

Στον ακόλουθο πίνακα μπορούμε να δούμε τον χρόνο εκτέλεσης σε msec, καθώς και το κόστος εκτέλεσης του βέλτιστου αλγορίθμου παράλληλης άθροισης με n δεδομένα εισόδου και με τη χρήση P επεξεργαστών. Επίσης, στον πίνακα φαίνεται ο χρόνος ($\log n$) που θεωρητικά πρέπει να έχει ο αλγόριθμος με βάση την ασυμπτωτικής ανάλυσης.

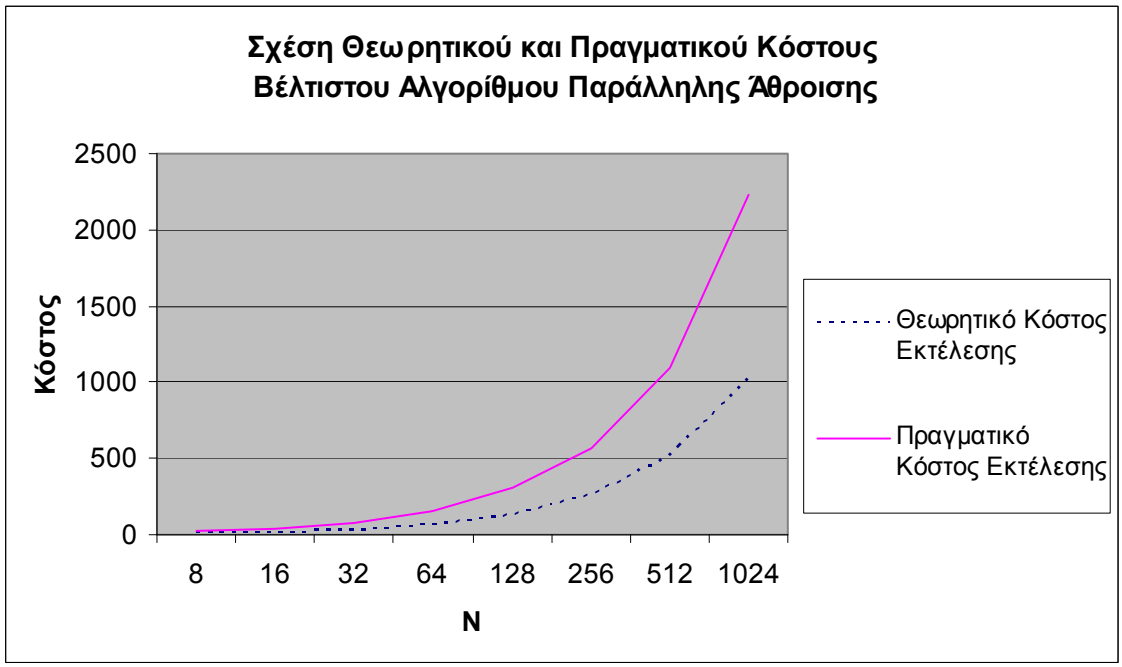
n	**P**	**T = $\log n$**	**Χρόνος**	**Κόστος**
8	3	3	8.44	25.32
16	4	4	9.77	39.08
32	7	5	11.37	79.59
64	11	6	13.74	151.14
128	19	7	16.35	310.65
256	32	8	17.73	567.36
512	57	9	19.27	1098.39
1024	103	10	21.7	2235.1

Πίνακας 4.2 – Μετρήσεις Προσομοίωσης

Στη συνέχεια ακολουθούν δύο γραφικές παραστάσεις, η Γραφική 4.3 στην οποία αναπαρίσταται ο χρόνος εκτέλεσης βάσει της θεωρητικής ανάλυσης μαζί με τον χρόνο που πήραμε από την προσομοίωση του βέλτιστου αλγορίθμου, και η Γραφική 4.4 στην οποία αναπαρίσταται το θεωρητικό και πραγματικό κόστος.



Γραφική 4.3 – Σχέση Θεωρητικού και Πραγματικού Χρόνου Εκτέλεσης



Γραφική 4.4 – Σχέση Θεωρητικού και Πραγματικού Κόστους Εκτέλεσης

Από τις μετρήσεις που πήραμε από την προσομοίωση του βέλτιστου αλγορίθμου παράλληλης άθροισης παρατηρούμε ότι όπως και προηγουμένως, ο χρόνος εκτέλεση αυξάνεται λογαριθμικά καθώς αυξάνονται τα δεδομένα εισόδου. Το γεγονός αυτό επιβεβαιώνει την εξάρτηση μεταξύ του χρόνου και του πλήθους των δεδομένων, αφού ο ασυμπτωτικός χρόνος εκτέλεσης δηλώνεται πάντα με βάση τα δεδομένα εισόδου και στην περίπτωση μας είναι ίσος με $\Theta(\log n)$. Ανάλογη σχέση με αυτή χρόνου – πλήθους δεδομένων, παρουσιάζεται και μεταξύ κόστους – πλήθους δεδομένων. Στην περίπτωση του κόστους όμως, όπως παρατηρούμε από την Γραφική 4.4 ότι η αύξηση είναι γραμμική. Άρα, επιβεβαιώνεται η θεωρητική ανάλυση βάσει την οποίας το κόστος εκτέλεσης του βέλτιστου αλγορίθμου είναι της τάξης του $\Theta(n)$.

Στην περίπτωση του βέλτιστου αλγόριθμου παρατηρούμε ότι το μέγιστο πλήθος των δεδομένων εισόδου φτάνει μέχρι $n = 1024$ και όχι $n = 2048$ όπως έγινε στον μη βέλτιστο αλγόριθμο παράλληλης άθροισης. Αυτό οφείλεται στο γεγονός ότι ο βέλτιστος αλγόριθμος είναι πιο πολύπλοκος και μετά από $n = 1024$ δεδομένα εισόδου παρουσιάζονται προβλήματα μνήμης που επηρεάζουν την ορθή λειτουργία του αλγόριθμου.

Από τις μετρήσεις αυτές μεγάλο ενδιαφέρων παρουσιάζει η σύγκριση μεταξύ του θεωρητικού χρόνου εκτέλεσης και του πραγματικού χρόνου εκτέλεσης. Οι τιμές χρόνου που λήφθηκαν από την προσομοίωση των αλγορίθμων διαφέρουν αρκετά από τις τιμές που δίνει η θεωρητική ανάλυση της πολυπλοκότητας χρόνου. Συγκεκριμένα, οι τιμές του πραγματικού χρόνου εκτέλεσης ξεπερνούν το διπλάσιο των τιμών του χρόνου που δίνει η θεωρητική αξιολόγηση, αλλά γενικά η διαφορά είναι κάποια σταθερά μεταξύ 5.50 και 11. Το γεγονός αυτό συμβαίνει διότι ο βέλτιστος αλγόριθμος αποτελείται από δύο μέρη, τον σειριακό υπολογισμό του αθροίσματος και την παράλληλη άθροιση. Τα δύο αυτά μέρη ξεχωριστά απαιτούν $\Theta(\log n)$ χρόνο εκτέλεσης. Ασυμπτωτικά και ο χρόνος εκτέλεσης ολόκληρου του αλγορίθμου είναι ίσος με $\Theta(\log n)$, αλλά στην πραγματικότητα τα δύο μέρη του αλγόριθμου επηρεάζουν και αυξάνουν τον πραγματικό χρόνο εκτέλεσης. Επομένως, η διαφορά μεταξύ του θεωρητικού χρόνου και του πραγματικού χρόνου εκτέλεσης του βέλτιστου αλγόριθμου παράλληλης άθροισης

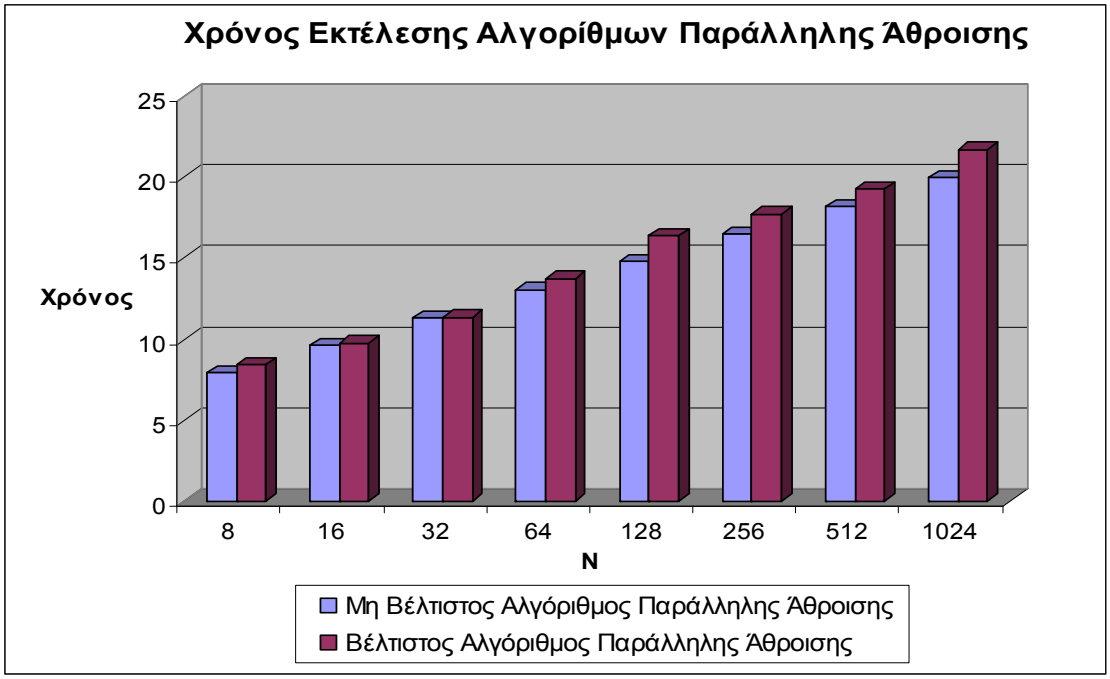
είναι δικαιολογημένη και οφείλεται στο ότι ο θεωρητικός χρόνος υπολογίζεται ασυμπτωτικά.

Στη συνέχεια, παρατηρούμε ότι και το κόστος αυξάνονται όταν αυξάνονται τα δεδομένα και ο χρόνος. Αυτό είναι φυσικό αφού το κόστος υπολογίζεται βάσει του πραγματικού χρόνου εκτέλεσης του αλγορίθμου. Αφού το κόστος εξαρτάται από τον χρόνο εκτέλεσης του αλγορίθμου, όσα παρατηρήθηκαν στην προηγούμενη παράγραφο για τη σχέση μεταξύ θεωρητικού και πραγματικού χρόνου εκτέλεσης του αλγορίθμου ισχύουν και για το κόστος.

4.6 Σύγκριση Αλγορίθμων

Σε αυτό το σημείο θα συγκρίνουμε το χρόνο και το κόστος που λήφθηκαν από την προσομοίωση των δύο αλγορίθμων. Με βάση όσα αναφέρθηκαν στις περιγραφές των αλγορίθμων, ο βέλτιστος αλγόριθμος πρέπει να είναι ο καλύτερος, δηλαδή ο πιο αποδοτικός. Απομένει μόνο να συγκρίνουμε τις μετρήσεις που πάρθηκαν για τους δύο αλγορίθμους και να συμπεράνουμε αν αυτό ισχύει στην πραγματικότητα.

Ακολουθεί μια γραφική παράσταση στην οποία παρουσιάζεται ο χρόνος εκτέλεσης των δύο αλγορίθμων παράλληλης άθροισης για τα διάφορα πλήθη δεδομένων εισόδου.



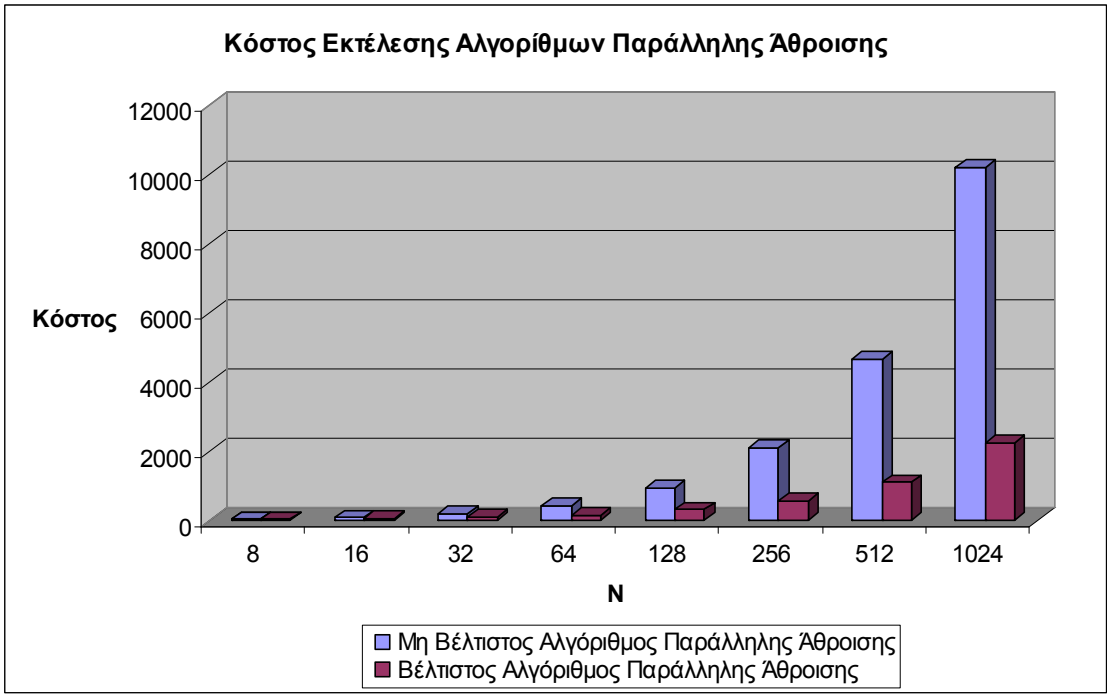
Γραφική 4.5 – Σύγκριση Αλγορίθμων Παράλληλης Άθροισης

Από την γραφική παράσταση παρατηρούμε ότι οι τιμές του χρόνου εκτέλεσης του βέλτιστου αλγόριθμου κυμαίνονται στο ίδια επίπεδα με τις τιμές του χρόνου εκτέλεσης του μη βέλτιστου αλγόριθμου. Παρόλα αυτά, ο μη βέλτιστος αλγόριθμος είναι λίγο πιο γρήγορος από τον βέλτιστο αλγόριθμο. Η διαφορά του χρόνου εκτέλεσης μεταξύ των δύο αλγορίθμων για μικρό αριθμό δεδομένων εισόδου είναι πολύ μικρή. Όσο όμως αυξάνονται τα δεδομένα εισόδου, αυξάνεται περισσότερο η διαφορά αυτή και ο μη βέλτιστος αλγόριθμος γίνεται ακόμα πιο γρήγορος σε σχέση με τον βέλτιστο αλγόριθμο.

Θεωρητικά, όμως οι δύο αλγόριθμοι έχουν τον ίδιο ασυμπτωτικό χρόνο εκτέλεσης που είναι $\Theta(\log n)$. Πρακτικά όμως παρατηρούμε μια μικρή διαφορά η οποία όχι μόνο δικαιολογείται αλλά είναι και απόλυτα φυσικό να παρουσιάζεται. Όπως αναφέραμε και προηγουμένως, ο χρόνος αυτός είναι ασυμπτωτικός οπόταν δεν λαμβάνει υπόψη του κάποιους υπολογισμούς. Ο βέλτιστος αλγόριθμος αποτελείται από δύο μέρη, τα οποία εκτελούνται ανεξάρτητα και χρειάζονται χρόνο $\Theta(\log n)$ το καθένα. Μάλιστα, ένα από τα δύο μέρη είναι ο μη βέλτιστος αλγόριθμος παράλληλης άθροισης. Από αυτό όμως κάποιος θα ανέμενε μεγαλύτερη διαφορά στον χρόνο εκτέλεσης του βέλτιστου αφού εκτελεί ολόκληρο τον μη βέλτιστο αλγόριθμο. Ο μη βέλτιστος αλγόριθμος όμως που εκτελείται ως μέρος του βέλτιστου αλγόριθμου λαμβάνει ως είσοδο λιγότερα δεδομένα εισόδου. Άρα, απαιτείται λιγότερος χρόνος για την εκτέλεση του σε σχέση με τον χρόνο που απαιτείται για την εκτέλεση του κανονικού μη βέλτιστου αλγορίθμου. Ως αποτέλεσμα όσον αναφέρθηκαν προηγουμένως, είναι λογικό ο μη βέλτιστος αλγόριθμος να είναι πιο γρήγορος από τον βέλτιστο, όχι όμως σε μεγάλο βαθμό.

Στη συνέχεια παρουσιάζεται η γραφική παράσταση με τα αποτελέσματα κόστους που συνεπάγονται των μετρήσεων που παρουσιάστηκαν σε προηγούμενο στάδιο. Θεωρητικά, το κόστος του βέλτιστου αλγόριθμου πρέπει να είναι καλύτερο από το κόστος του μη βέλτιστου. Αν και οι δύο αλγόριθμου έχουν ασυμπτωτικά τον ίδιο χρόνο εκτέλεσης, το κόστος του βέλτιστου είναι καλύτερο αφού χρησιμοποιεί λιγότερους επεξεργαστές. Συγκεκριμένα, χρησιμοποιεί $n \,/\, \log n$ επεξεργαστές και άρα το κόστος του είναι $\Theta(n)$ σε αντίθεση με τον μη βέλτιστο αλγόριθμο που έχει κόστος $\Theta(n \log n)$.

Άρα, θεωρητικά ο βέλτιστος αλγόριθμος είναι πιο αποδοτικός σε θέματα κόστους και το ερώτημα είναι αν κάτι τέτοιο ισχύει και στην πραγματικότητα.



Γραφική 4.6 – Σύγκριση Αλγορίθμων Παράλληλης Άθροισης

Παρατηρώντας την πιο πάνω γραφική παράσταση παρατηρούμε μια διαφορά μεταξύ του κόστους των δύο αλγορίθμων. Ο βέλτιστος αλγόριθμος έχει μικρότερο κόστος εκτέλεσης σε σχέση με τον μη βέλτιστο. Επιπλέον, όσο ο αριθμός των δεδομένων εισόδου αυξάνονται, τόσο περισσότερο αυξάνεται η διαφορά κόστους μεταξύ των δύο αλγορίθμων. Αυτό είναι λογικό αφού όπως προαναφέραμε, ασυμπτωτικά ο βέλτιστος αλγόριθμος είναι καλύτερος από τον μη βέλτιστο σε θέματα κόστους.

Στον πίνακα που ακολουθεί μπορούμε να παρατηρήσουμε τις διαφορές που αφορούν την πολυπλοκότητα κόστους μεταξύ του βέλτιστου και του μη βέλτιστού αλγορίθμου.

n	**Βέλτιστος Αλγόριθμος**	**Μη Βέλτιστος Αλγόριθμος**	**Διαφορά Κόστους**
8	25.32	31.68	6.36
128	310.65	944.64	633.99
1024	2235.1	10234.88	7999.78

Πίνακας 4.3 – Διαφορές Κόστους Αλγορίθμων

Όπως αναφέραμε και προηγουμένως, όταν ο αριθμός των δεδομένων αυξάνεται, τότε αυξάνεται και η διαφορά κόστους μεταξύ των δύο αλγορίθμων. Παρατηρούμε ότι για τον μικρότερο αριθμό δεδομένων εισόδου η διαφορά είναι μικρή και για μεσαίο αριθμό δεδομένων εισόδου το κόστος του βέλτιστου είναι το μισό από αυτό του μη βέλτιστου. Για μεγάλο αριθμό δεδομένων εισόδου η διαφορά του κόστους γίνεται κατά πολύ μεγαλύτερη σε σχέση με μικρό πλήθος δεδομένων.

Άρα, συμπεραίνουμε ότι ο βέλτιστος αλγόριθμος είναι ιδιαίτερα αποδοτικός σε περιπτώσεις που έχουμε μεγάλα δεδομένα εισόδου. Κάτι τέτοιο είναι πολύ θετικό αφού σημαίνει ότι ο αλγόριθμος αυτός θα ανταποκρίνεται ικανοποιητικά σε θέμα κόστους για ρεαλιστικά προβλήματα με μεγάλο αριθμό δεδομένων εισόδου.

Για να συμπεράνουμε αν ο βέλτιστος αλγόριθμος είναι όντως καλύτερος από τον μη βέλτιστο, θα συγκρίνουμε τη διαφορά χρόνου εκτέλεσης των δύο αλγορίθμων. Στον πίνακα που ακολουθεί μπορούμε να παρατηρήσουμε τις διαφορές που αφορούν την πολυπλοκότητα χρόνου μεταξύ του βέλτιστου και του μη βέλτιστού αλγορίθμου παράλληλης άθροισης.

n	**Βέλτιστος Αλγόριθμος**	**Μη Βέλτιστος Αλγόριθμος**	**Διαφορά Χρόνου**
8	8.44	7.92	0.52
128	16.35	14.76	1.59
1024	21.70	19.99	1.71

Πίνακας 4.4 – Διαφορές Χρόνου Αλγορίθμων

Από τον πίνακα παρατηρούμε ότι η διαφορά χρόνου μεταξύ των δύο αλγορίθμων δεν είναι πολύ μεγάλη, αλλά αντίθετα είναι ελάχιστη. Αν και αυξάνεται με την αύξηση των δεδομένων εισόδου, η αύξηση αυτή είναι πολύ μικρή. Στον ελάχιστο αριθμό δεδομένων εισόδου η διαφορά του χρόνου εκτέλεσης είναι 0.52, ενώ στο μεγαλύτερο αριθμό δεδομένων η διαφορά φτάνει μόνο το 1.71 . Άρα, η διαφορά χρόνου παραμένει σχεδόν σταθερή για τα διάφορα δεδομένα εισόδου.

Επομένως, η διαφορά που παρουσιάζεται στον χρόνο εκτέλεσης των δύο αλγορίθμων είναι πολύ μικρή και άρα ασήμαντη. Έτσι, μπορούμε να πούμε ότι και στην πραγματικότητα ο χρόνος εκτέλεσης των δύο αλγορίθμων είναι παρόμοιος.

Με βάση την ανάλυση και την αξιολόγηση των μετρήσεων που λήφθηκαν από την προσομοίωση των αλγορίθμων συμπεραίνουμε ότι ο βέλτιστος αλγόριθμος παράλληλης άθροισης είναι ο πιο αποδοτικός. Αν και οι δύο αλγόριθμοι έχουν παρόμοιο χρόνο εκτέλεσης, η πολυπλοκότητα κόστους στον βέλτιστο αλγόριθμο είναι κατά πολύ μικρότερη από αυτή του μη βέλτιστου αλγόριθμου.

Κεφάλαιο 5

Αλγόριθμος Παράλληλης Μετάδοσης

5.1 Πρόβλημα

Το πρόβλημα της μετάδοσης στο PRAM αφορά μια τιμή που βρίσκεται σε μια θέση της κοινόχρηστης μνήμης. Θεωρούμε ότι η τιμή βρίσκεται στη πρώτη θέση $SM(1)$ της κοινόχρηστης μνήμης και επιθυμούμε η τιμή αυτή να μεταδοθεί, δηλαδή να αντιγραφεί σε n συνεχόμενες θέσεις της κοινόχρηστης μνήμης.

5.2 Σειριακός Αλγόριθμος

Όταν έχουμε στη διάθεση μας ένα επεξεργαστή η λύση είναι απλή. Με μια επανάληψη, ο επεξεργαστής διαβάζει κάθε φορά την τιμή στη θέση 1 και μετά την αποθηκεύει σε μια άλλη θέση, μέχρι η τιμή να αντιγραφεί στις n επιθυμούμενες θέσεις.

Σειριακός Αλγόριθμος:

$for\ j = 2\ to\ n\ do$
    $S[j] = S[1]$
$end\ do$

Όπως παρατηρούμε, η επανάληψη στην οποία γίνεται η αντιγραφή της αρχικής τιμής στις υπόλοιπες θέσεις εκτελείται $n-1$ φορές, δηλαδή είναι της τάξης $\Theta(n)$. Είναι λογικό να χρειάζεται τέτοιος χρόνος αφού η τιμή πρέπει να αντιγραφεί σε n θέσεις και

αυτό μπορεί να γίνει μόνο αν ο επεξεργαστής εκτελέσει την αντιγραφή της τιμής για κάθε θέση της μνήμης ξεχωριστά. Το περιεχόμενο της επανάληψης, δηλαδή η πράξη με την οποία γίνεται η μετάδοση εκτελείται σε σταθερό χρόνο $\Theta(1)$.

Άρα, ο συνολικός χρόνος εκτέλεσης του σειριακού αλγόριθμου μετάδοσης είναι $\Theta(n)$. Στη περίπτωση όμως που έχουμε περισσότερους από ένα επεξεργαστές, ο χρόνος εκτέλεσης θεωρητικά είναι μικρότερος αφού υπάρχει η δυνατότητα παράλληλης εκτέλεσης της αντιγραφής της τιμής.

5.3 Παράλληλος Αλγόριθμος

Στην περίπτωση που η μετάδοση μιας τιμής μπορεί να εκτελεστεί στα μοντέλα CREW και CRCW PRAM, η λύση του προβλήματος είναι πολύ απλή. Στα μοντέλα στα οποία επιτρέπεται ταυτόχρονη ανάγνωση της ίδιας κυψελίδας της κοινόχρηστης μνήμης από περισσότερους από ένα επεξεργαστές, αρκεί ο κάθε επεξεργαστής να διαβάσει την πρώτη κυψελίδα μνήμης $SM(1)$ στην οποία βρίσκεται η τιμή μετάδοσης και να την αποθηκεύσει στη θέση μνήμης με που έχει ίδιο αναγνωριστικό με τον συγκεκριμένο επεξεργαστή. Αυτή η διαδικασία γίνεται παράλληλα από $n-1$ επεξεργαστές, όσες είναι και οι θέσεις στις οποίες θέλουμε να μεταδώσουμε την τιμή $SM(1)$.

$Processors\ i = 2\ to\ n\ do\ in\ parallel$
    $SM[i] = SM[1]$
$end\ do$

Είναι προφανές ότι η εκτέλεση της μετάδοσης στα μοντέλα CREW και CRCW PRAM απαιτεί σταθερό χρόνο $\Theta(1)$. Το κόστος της μετάδοσης είναι $\Theta(n)$ αφού χρησιμοποιούνται $n-1$ επεξεργαστές.

Στην περίπτωση όμως του μοντέλου EREW PRAM, όπου δεν επιτρέπεται η ταυτόχρονη ανάγνωση η επίλυση του προβλήματος της μετάδοσης δεν είναι τόσο απλή. Θα πρέπει με κάποιο τρόπο οι επεξεργαστές που λειτουργούν παράλληλα να μεταδίδουν την τιμή στη θέση $SM[1]$ χωρίς να έχουν πρόσβαση όλοι μαζί στην κυψελίδα αυτή.

Ο τρόπος επίλυσης του προβλήματος αυτού στηρίζεται σε μέθοδο παρόμοια και αντίστροφη με αυτή που χρησιμοποιήθηκε στον αλγόριθμο παράλληλης άθροισης. Στη συγκεκριμένη περίπτωση ξεκινά η εκτέλεση του αλγορίθμου με τη χρήση ενός επεξεργαστή και στη συνέχεια οι επεξεργαστές διπλασιάζονται μέχρι να φτάσουν τον αριθμό $n/2$. Άρα, ο μέγιστος αριθμός επεξεργαστών που χρειάζονται για την επίλυση του προβλήματος της μετάδοσης στο μοντέλο EREW PRAM είναι $n/2$.

Αρχικά, ο πρώτος επεξεργαστής μόνο διαβάζει την κυψελίδα $SM[1]$ και την αντιγράφει στη δεύτερη κυψελίδα της κοινόχρηστης μνήμης. Έτσι, σε αυτό το σημεία υπάρχουν δύο ίδιες τιμές στην κοινόχρηστη μνήμη. Έπειτα, είναι σε λειτουργία δύο επεξεργαστές οι οποίοι διαβάζουν από μία από τις κυψελίδα μνήμης που έχουν την τιμή και την αντιγράφουν στις επόμενες δύο άδειες κυψελίδες μνήμης. Η διαδικασία επαναλαμβάνεται μέχρι να αντιγραφή η αρχική τιμή και στις n κυψελίδες της κοινόχρηστης μνήμης. Κάθε φορά οι επεξεργαστές διπλασιάζονται και ο καθένας διαβάζει μια από τις τιμές που γράφτηκαν ήδη στη κοινόχρηστη μνήμη. Συγκεκριμένα ο κάθε επεξεργαστής την τιμή που βρίσκονται σε θέση ίδια με το αναγνωριστικό του επεξεργαστή. Έπειτα, ο κάθε επεξεργαστής γράφει την τιμή που διάβασε σε επόμενη κενή θέση της κοινόχρηστης μνήμης ανάλογα με το αναγνωριστικό του. Για παράδειγμα, ο επεξεργαστής P_1 γράφει στην πρώτη κενή θέση, ο P_2 στη δεύτερη και ο P_n στην $n^{οστή}$.

Ακολουθεί ο αλγόριθμος παράλληλης μετάδοσης [5] :

$Processors\ i = 1\ to\ n\ do\ in\ parallel$
$j = 0$
$while\ (2^j < n)$
    $if\ (i \leq 2^j)$
        $SM[i + 2^j] = SM[i]$
    $end\ if$
    $j = j + 1$
$end\ while$
$end\ do$

Ο προηγούμενος αλγόριθμος εκτελείται σε χρόνο $\Theta(\log n)$, εξαιτίας του αριθμού των επαναλήψεων. Όπως αναφέρθηκε και προηγουμένως, σε κάθε επανάληψη ο αριθμός των τιμών που αντιγράφονται στις κυψελίδες της κοινόχρηστης μνήμης διπλασιάζονται για αυτό και ο χρόνος εκτέλεσης της επανάληψης είναι $\Theta(\log n)$. Οι πράξεις τόσο στο εξωτερικό όσο και στο εξωτερικό της επανάληψης απαιτούν σταθερό χρόνο για την εκτέλεση τους. Έτσι, ο συνολικός χρόνος εκτέλεσης του αλγορίθμου αυτού είναι $\Theta(\log n)$.

Στον αλγόριθμο αυτό λειτουργούν το πολύ $n/2$ επεξεργαστές, δηλαδή ο αριθμός των επεξεργαστών που χρειαζόμαστε για την εκτέλεση του αλγορίθμου αυτού είναι της τάξης $\Theta(n)$. Αφού όπως διαπιστώθηκε στη προηγούμενη παράγραφο ο χρόνος εκτέλεσης του αλγορίθμου είναι $\Theta(\log n)$, το κόστος για την εκτέλεση του είναι $\Theta(n)$*$\Theta(\log n)$. Άρα, η πολυπλοκότητα κόστους του παράλληλου αλγορίθμου μετάδοσης στο μοντέλο EREW PRAM είναι $\Theta(n \log n)$.

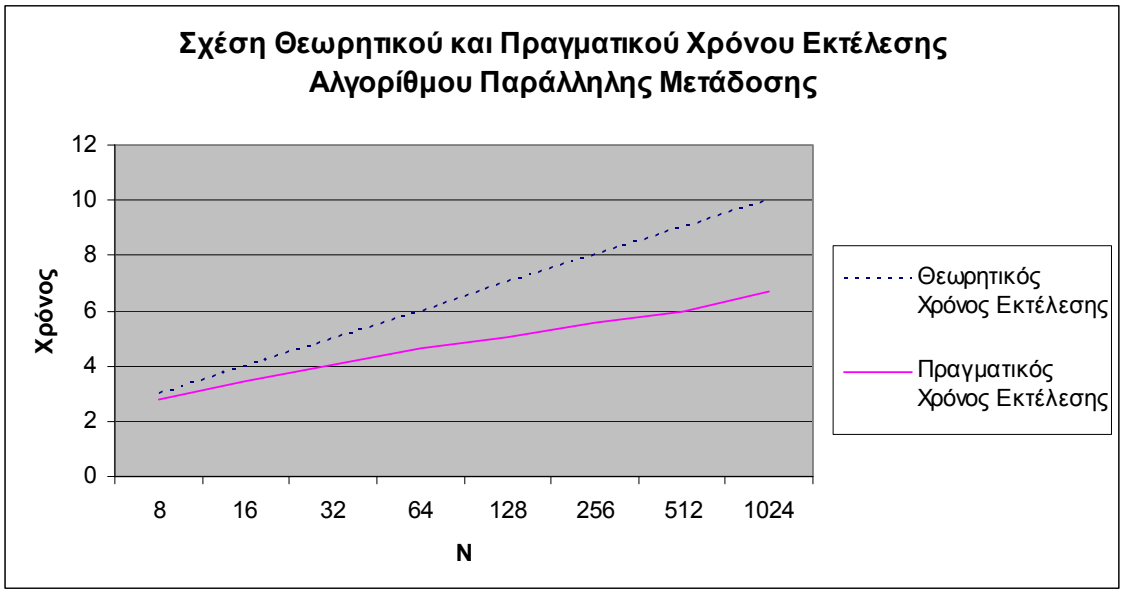
5.4 Πειραματική Αξιολόγηση Αλγορίθμου

Ο παράλληλος αλγόριθμος μετάδοσης μιας τιμής που υλοποιήθηκε για το μοντέλο EREW PRAM, αναπτύχθηκε με την χρήση της γλώσσας XMTC και προσομοιώθηκε στον προσομοιωτή XMT για να πάρουμε τα αποτελέσματα. Πιο κάτω φαίνονται ο μετρήσεις που πάρθηκαν από την προσομοίωση, για αξιολόγηση του παράλληλου αλγορίθμου. Οι μετρήσεις πάρθηκαν από την εκτέλεση του αλγορίθμου με διαφορετικό αριθμό δεδομένων εισόδου n κάθε φορά. Στον ακόλουθο πίνακα μπορούμε να δούμε τον χρόνο (msec) και το κόστος εκτέλεσης του αλγορίθμου με n δεδομένα εισόδου και με τη χρήση P επεξεργαστών. Επίσης, στον πίνακα φαίνεται ο χρόνος $(\log n)$ που θεωρητικά πρέπει να έχει ο αλγόριθμος βάση της ασυμπτωτικής ανάλυσης.

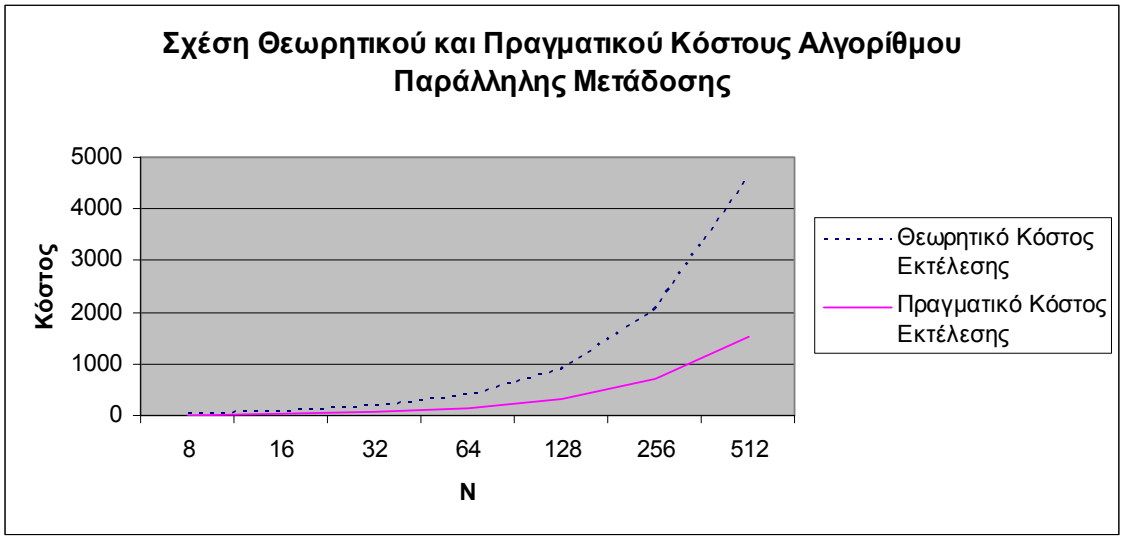
n	P	$\log n$	Χρόνος	Κόστος
8	4	3	2.77	11.08
16	8	4	3.44	27.52
32	16	5	4.06	64.96
64	32	6	4.62	147.84
128	64	7	5.02	321.28
256	128	8	5.58	714.24
512	256	9	5.99	1533.44
1024	512	10	6.70	3430.40

Πίνακας 5.1 – Μετρήσεις Προσομοίωσης

Στη συνέχεια ακολουθούν δύο γραφικές παραστάσεις, η Γραφική 5.1 στην οποία αναπαρίσταται ο χρόνος εκτέλεσης βάσει της θεωρητικής ανάλυσης μαζί με τον χρόνο που πήραμε από την προσομοίωση του αλγορίθμου παράλληλης μετάδοσης, και η Γραφική 5.2 στην οποία αναπαρίσταται το θεωρητικό και πραγματικό κόστος.



Γραφική 5.1 – Σχέση Θεωρητικού και Πραγματικού Χρόνου Εκτέλεσης



Γραφική 5.2 – Σχέση Θεωρητικού και Πραγματικού Κόστους Εκτέλεσης

Από τις μετρήσεις συμπεραίνουμε ότι ο χρόνος εκτέλεσης του αλγορίθμου εξαρτάται από τον αριθμό των δεδομένων εισόδου. Από την Γραφική 5.1 παρατηρούμε ότι όταν ο αριθμός των δεδομένων αυξάνεται, τότε αυξάνεται λογαριθμικά ο χρόνος που χρειάζεται για την ολοκλήρωση του αλγόριθμου. Το γεγονός αυτό επιβεβαιώνει την θεωρητική ανάλυση βάσει της οποίας ο χρόνος εκτέλεσης είναι της τάξης $\Theta(\log n)$.

Το κόστος υπολογίζεται βάση του αριθμού των δεδομένων εισόδου και του χρόνου εκτέλεσης. Άρα, και το κόστος εξαρτάται από το πλήθος των δεδομένων εισόδου. Στην περίπτωση του κόστους όμως η αύξηση που παρατηρείται είναι της μορφής $n \log n$. Αυτό είναι αναμενόμενο, αφού το κόστος εκτέλεσης ασυμπτωτικά δηλώνεται βάσει του πλήθους των δεδομένων εισόδου και είναι της τάξης $\Theta(n \log n)$.

Ενδιαφέρον είναι να προσέξουμε τη διαφορά μεταξύ του θεωρητικού χρόνου εκτέλεσης του αλγορίθμου και του πραγματικού χρόνου στον οποίο εκτελείται ο αλγόριθμος. Σε αυτό τον αλγόριθμο παρατηρούμε να συμβαίνει το εξής παράξενο γεγονός: Ο θεωρητικός χρόνος εκτέλεσης του αλγορίθμου είναι μεγαλύτερος από τον πραγματικό χρόνο που χρειάζεται για την εκτέλεση του. Αυτό συμβαίνει εξαιτίας της απλότητας του προβλήματος, και του γεγονότος ότι στο εσωτερικό των επαναλήψεων δεν εκτελούνται πάντα όλες οι εντολές. Παρόλα αυτά οι δύο χρόνοι δεν έχουν μεγάλη διαφορά μεταξύ τους. Το ανάλογο συμβαίνει και για το κόστος, αφού είναι πλέον γνωστό ότι το κόστος εξαρτάται από τον χρόνο εκτέλεσης.

Κεφάλαιο 6

Παράλληλος Αλγόριθμος Προθεματικού Αθροίσματος

6.1 Πρόβλημα

Το πρόβλημα του προθεματικού αθροίσματος περιλαμβάνει μια λίστα από n αριθμούς x_1, x_2, …, x_n οι οποίοι αποτελούν τα δεδομένα του προβλήματος. Πρέπει με κάποιο αποτελεσματικό τρόπο να υπολογίσουμε τα προθεματικά αθροίσματα των αριθμών αυτών. Ο αλγόριθμος δηλαδή, δεν υπολογίζει μόνο το τελικό άθροισμα όλων των αριθμών της λίστας, αλλά και τα ενδιάμεσα αθροίσματα σε προθέματα. Ο αριθμός των προθεμάτων είναι ίδιος με το πλήθος των δεδομένων εισόδου. Συγκεκριμένα, στην περίπτωση που έχουμε n δεδομένα εισόδου θα υπολογιστούν τα ακόλουθα προθέματα: $\pi_1 = x_1,\ \pi_2 = x_1 + x_2,\ \pi_3 = x_1 + x_2 + x_3, \ldots,\ \pi_n = x_1 + x_2 + x_3 + \ldots + x_n$.

6.2 Σειριακός Αλγόριθμος

Η σειριακή λύση του προβλήματος προθεματικού αθροίσματος, στην οποία χρησιμοποιείται ένας μόνο επεξεργαστής είναι παρόμοια με τη σειριακή λύση του προβλήματος του αθροίσματος. Η μόνη διαφορά μεταξύ των δύο αλγορίθμων είναι ότι

το άθροισμα που υπολογίζεται δεν αποθηκεύεται σε μια απλή μεταβλητή αλλά σε πίνακα. Με αυτό τον τρόπο, αποθηκεύουμε σε κάθε θέση του πίνακα το άθροισμα των αριθμών που προστέθηκαν μεταξύ τους μέχρι τώρα. Δηλαδή, σε κάθε θέση του πίνακα αποθηκεύεται το πρόθεμα των αριθμών μέχρι τη θέση αυτή.

Ο σειριακός αλγόριθμος για το προθεματικό άθροισμα είναι ο ακόλουθος [5]:

$\pi_1 = x_1$
for $i = 1$ *to* $n-1$ *do*
    $\pi_{i+1} = \pi_i + x_{i+1}$
end do

Ο αλγόριθμος εκτελείται σε χρόνο $\Theta(n)$, αφού ο ένας επεξεργαστής που έχουμε στη διάθεσή μας εκτελεί μια επανάληψη $n-1$ φορές για να κάνει την άθροιση των στοιχείων της λίστας. Αρχικά, διαβάζει σε σταθερό χρόνο $\Theta(1)$ το πρώτο στοιχείο και καθορίζει το στοιχείο αυτό ως το πρώτο πρόθεμα. Έπειτα, υπολογίζονται τα υπόλοιπα $n-1$ προθέματα και αποθηκεύονται στις αντίστοιχες θέσεις του πίνακα.

Άρα, ο συνολικός χρόνος εκτέλεσης του σειριακού αλγορίθμου προθεματικού αθροίσματος είναι $\Theta(n)$. Στη περίπτωση όμως που μπορούμε να έχουμε περισσότερους από ένα επεξεργαστές, ο χρόνος εκτέλεσης θεωρητικά είναι μικρότερος αφού υπάρχει η δυνατότητα παράλληλης εκτέλεσης των πράξεων.

6.3 Πρώτος Παράλληλος Αλγόριθμος EREW

Στην περίπτωση που το πρόβλημα του προθεματικού αθροίσματος επιλύεται με παράλληλο αλγόριθμο στο μοντέλο EREW χρησιμοποιούνται $n-1$ επεξεργαστές και έχουμε στη διάθεση μας n στοιχεία. Τα στοιχεία αυτά, δηλαδή τα δεδομένα του προβλήματος είναι αποθηκευμένα σε ένα πίνακα στην κοινόχρηστη μνήμη.

Ο αλγόριθμος υλοποιείται βάσει της τεχνικής του αναδρομικού διπλασιασμού. Η τεχνική αυτή αρχικά συνδυάζει ζευγάρια στοιχείων, και στη συνέχεια συνδυάζει ζευγάρια ζευγαριών μέχρι την επίλυση ενός δεδομένου προβλήματος.

Χρησιμοποιώντας την τεχνική αυτή ο κάθε ένας από τους $n-1$ επεξεργαστές υπολογίζει και ένα πρόθεμα, ανάλογα με το αναγνωριστικό του.

Αρχικά, όλα τα προθέματα έχουν την τιμή των στοιχείων που βρίσκονται στη θέση του πίνακα με τιμή ίδια με το αναγνωριστικό του επεξεργαστή που εκτελεί την πράξη. Έπειτα, για τον υπολογισμό των προθεμάτων όλοι οι επεξεργαστές αθροίζουν το i στοιχείο από τον πίνακα στην κοινόχρηστη μνήμη με το προηγούμενο από αυτό στοιχείο. Στη συνέχεια, σε κάθε επανάληψη ο κάθε επεξεργαστής αθροίζει την τιμή που έχει υπολογίσει προηγουμένως με την τιμή που καθορίζεται από την τιμή $i-s$.

Η βοηθητική μεταβλητή s, διπλασιάζεται σε κάθε επανάληψη ώστε να λειτουργούν μόνο οι επεξεργαστές με αναγνωριστικό μεγαλύτερο από την τιμή της μεταβλητής s. Στην πρώτη εκτέλεση της επανάληψης λειτουργούν και κάνουν υπολογισμούς όλοι οι επεξεργαστές. Άρα, για αυτό τον αλγόριθμο χρησιμοποιούμε $n-1$ επεξεργαστές στη χειρότερη περίπτωση.

Ο αλγόριθμος EREW για το προθεματικό άθροισμα είναι [5]:

Processors i = 2 to n do in parallel
    $s = 1$
    while $s < i$
        $\pi[i] = \pi[i-s] + \pi[i]$
        $s = 2 * s$
    end while
end do

Το μοντέλο στο οποίο υλοποιείται ο αλγόριθμος αυτός είναι EREW, αφού δεν υπάρχει ταυτόχρονη ανάγνωση ή γραφή στην ίδια κυψελίδα κοινόχρηστης μνήμης από περισσότερους από ένα επεξεργαστές. Όταν απαιτείται ανάγνωση από την κοινόχρηστη μνήμη, γίνεται βάση του μοναδικού αναγνωριστικού του κάθε επεξεργαστή. Τι ίδιο ισχύει και για την γραφή, όπου η αποθήκευση στη κοινόχρηστη μνήμη γίνεται στη θέση του πίνακα που είναι ίδια με το αναγνωριστικό του επεξεργαστή που εκτελεί την πράξη.

Ο ασυμπτωτικός χρόνος εκτέλεσης του αλγορίθμου αυτού είναι $\Theta(\log n)$. Όπως παρατηρούμε στο πιο πάνω ψευδοκώδικα, ο χρόνος αυτός οφείλεται στην εκτέλεση των επαναλήψεων του βρόγχου αφού εκτελείται σε χρόνο της τάξης του $\Theta(\log n)$. Οι πράξεις στο εσωτερικό του βρόγχου απαιτούν σταθερό χρόνο, δηλαδή είναι της τάξης $\Theta(1)$. Μετά από κάθε επανάληψη μειώνεται στο μισό ο αριθμός των επεξεργαστών που είναι σε λειτουργία, εξαιτίας της συνθήκης $s < i$ του βρόγχου. Αυτό οφείλεται στο ότι η μεταβλητή s διπλασιάζεται σε κάθε επανάληψη.

Αφού ο χρόνος εκτέλεσης του αλγορίθμου είναι $\Theta(\log n)$ και χρησιμοποιούνται n επεξεργαστές για τους υπολογισμούς, τότε το κόστος είναι $\Theta(n)$*$\Theta(\log n)$. Άρα, το τελικό κόστος εκτέλεσης του αλγόριθμου παράλληλου προθεματικού υπολογισμού στο μοντέλο EREW είναι $\Theta(n \log n)$.

6.4 Δεύτερος Παράλληλος Αλγόριθμος CREW

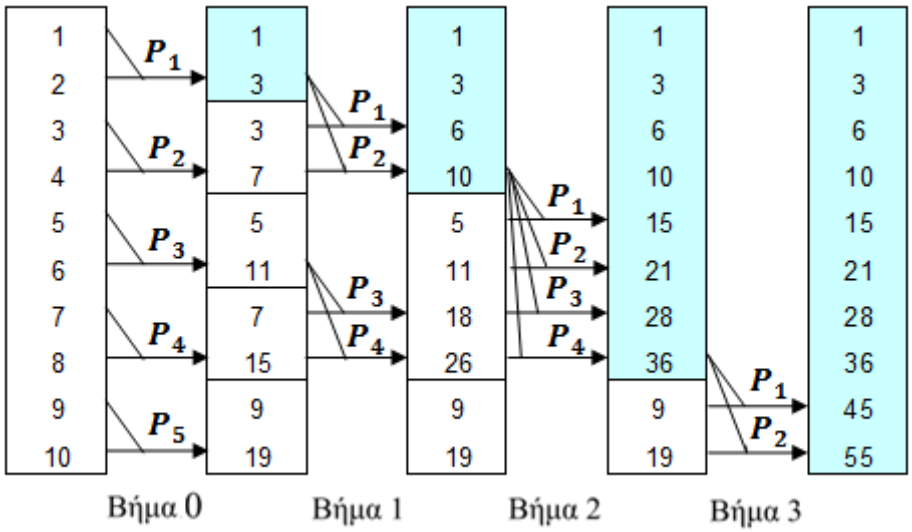
Σε αυτό τον αλγόριθμο χρησιμοποιούνται $n/2$ επεξεργαστές για την επίλυση του προβλήματος του προθεματικού αθροίσματος, με n στοιχεία εισόδου, αποθηκευμένα σε πίνακα στη κοινόχρηστη μνήμη.

Ο αλγόριθμος αυτός, όπως και ο προηγούμενος, βασίζεται στη τεχνική του αναδρομικού διπλασιασμού. Υπάρχει όμως διαφορά στα ζευγάρια προθεμάτων τα οποία συνδυάζονται για την επίλυση του προβλήματος. Για την υλοποίηση του αλγορίθμου αυτού απαιτείται ταυτόχρονο διάβασμα της ίδιας κυψελίδας της κοινόχρηστης μνήμης από περισσότερους από ένα επεξεργαστές, δηλαδή υλοποιείται στο μοντέλο CREW.

Αρχικά, ο κάθε επεξεργαστής υπολογίζει το άθροισμα δύο στοιχείων, τα οποία βρίσκονται σε συνεχόμενες θέσεις στη κοινόχρηστη μνήμη (βήμα 0). Τα αθροίσματα αυτά αποτελούν τα πρώτα προθέματα που υπολογίζονται, τα οποία όμως θα μετέχουν σε υπολογισμούς στη συνέχεια. Έπειτα, συνεχίζονται οι υπολογισμοί συνδυάζοντας διαφορετικά ζευγάρια στοιχείων σε κάθε επανάληψη. Ο κάθε επεξεργαστής αθροίζει ένα από τα προθέματα που υπολογίστηκαν σε προηγούμενο βήμα με ένα άλλο στοιχείο. Εκτός από το βήμα 0, σε όλες τις επόμενες επαναλήψεις υπάρχουν περισσότεροι από

ένας επεξεργαστής που διαβάζουν την ίδια κυψελίδα μνήμης. Η κυψελίδα αυτή είναι η κυψελίδα στην οποία βρίσκεται αποθηκευμένο το πρόθεμα που υπολογίστηκε σε κάποιο προηγούμενο βήμα και χρησιμοποιείται για να αθροιστεί με ένα άλλο στοιχείο, διαφορετικό για τον κάθε επεξεργαστή. Σε κάθε επανάληψη διπλασιάζεται ο αριθμός των επεξεργαστών που έχουν ταυτόχρονη πρόσβαση στην ίδια κυψελίδα μνήμης.

Ακολουθεί ένα παράδειγμα που περιγράφει σχηματικά τον τρόπο λειτουργίας του αλγορίθμου αυτού. Στο παράδειγμα υπάρχουν δέκα στοιχεία εισόδου και άρα χρησιμοποιούνται πέντε επεξεργαστές για τον υπολογισμό των προθεμάτων.



Σχήμα 6.1 – Παράδειγμα Εκτέλεσης Αλγορίθμου

Ο παράλληλος αλγόριθμος προθεματικού αθροίσματος είναι ο πιο κάτω:

$Processors\ i = 1\ to\ n/2\ do\ in\ parallel$

$j = 0$

$SM[2i] = SM[2i] + SM[2i-1]$

$while\ (n > 2^{j+1})$

$position =< next\ list\ position >$

$SM[position] = SM[position] + SM[position - s]$

$j = j + 1$

$end\ while$

$end\ do$

Στον προηγούμενο ψευδοκώδικα για τον προθεματικό αλγόριθμο, η μεταβλητή $position$ καθορίζει τη θέση του κοινόχρηστου πίνακα στην οποία πρέπει να αποθηκευτεί το αποτέλεσμα των πράξεων σε κάθε βήμα.

Ο χρόνος εκτέλεσης του αλγορίθμου αυτού είναι $\Theta(\log n)$. Οι πράξεις εκτός της επανάληψης όπου διαβάζονται δύο αριθμοί και υπολογίζεται το αρχικό άθροισμα μεταξύ των αριθμών αυτών της κοινόχρηστης μνήμης χρειάζονται σταθερό χρόνο. Η εκτέλεση των πράξεων στο εσωτερικό της επανάληψης απαιτούν επίσης σταθερό χρόνο, αφού και εδώ απλά διαβάζονται στοιχεία από την μνήμη και υπολογίζεται το άθροισμα τους. Οι πράξεις αυτές του εσωτερικού της επανάληψης εκτελούνται $\Theta(\log n)$ φορές, εξαιτίας της συνθήκης $\left(n > 2^{j+1}\right)$. Συγκεκριμένα, η συνθήκη αυτή καθορίζει πόσες φορές θα χρειαστεί να αθροίσει ζεύγη στοιχείων ο κάθε επεξεργαστής μέχρι να βρεθεί λύση. Αφού χρησιμοποιείται η τεχνική του αναδρομικού διπλασιασμού, όπου συνδυάζονται συνεχώς ζευγάρια μέχρι την εύρεση λύσης, ο χρόνος εκτέλεσης των επαναλήψεων, άρα και ολόκληρου του αλγόριθμου είναι $\Theta(\log n)$ [4].

Για την επιτυχημένη εκτέλεση του αλγορίθμου είναι απαραίτητη η χρήση $n/2$ επεξεργαστών, δηλαδή χρησιμοποιούνται $\Theta(n)$ επεξεργαστές. Αφού ο χρόνος εκτέλεσης του παράλληλου αλγορίθμου είναι $\Theta(\log n)$, το κόστος είναι $\Theta(n)$ *$\Theta(\log n)$. Δηλαδή, η πολυπλοκότητα όσο αφορά το κόστος του παράλληλου αλγορίθμου προθεματικού αθροίσματος στο μοντέλο CREW PRAM είναι της τάξης $\Theta(n \log n)$.

Μετά από την θεωρητική ανάλυση της πολυπλοκότητας των δύο παράλληλων αλγορίθμων προθεματικού αθροίσματος παρατηρούμε ότι τόσο στο μοντέλο EREW PRAM, όσο και στο μοντέλο CREW PRAM οι αλγόριθμοι έχουν το ίδιο κόστος και χρόνο εκτέλεσης. Συγκεκριμένα, θεωρητικά χρειάζεται χρόνος $\Theta(\log n)$ και κόστος $\Theta(n \log n)$ για την επιτυχημένη εκτέλεση του κάθε αλγόριθμου. Συμπεραίνουμε όμως ότι κανένας από τους δύο αλγόριθμους δεν είναι βέλτιστος αφού ο βέλτιστος σειριακός αλγόριθμος που λύνει το πρόβλημα αυτό έχει πολυπλοκότητα της τάξης $\Theta(n)$ που είναι μικρότερη από το κόστος των παράλληλων αλγορίθμων.

Στόχος μας είναι η υλοποίηση ενός παράλληλου αλγορίθμου που να εκτελεί το προθεματικό άθροισμα με κόστος $\Theta(n)$, δηλαδή η εύρεση ενός βέλτιστου αλγόριθμου που να επιλύει το πρόβλημα αυτό.

6.5 Βέλτιστος Αλγόριθμος

Για να πετύχουμε την υλοποίηση ενός βέλτιστου αλγορίθμου πρέπει να αναπτύξουμε αλγόριθμο με κόστος ίσο με την πολυπλοκότητα του καλύτερου σειριακού αλγορίθμου. Στην περίπτωσή μας θέλουμε να έχουμε κόστος $\Theta(n)$ αντί $\Theta(n \log n)$ που είναι το κόστος των αλγορίθμων που περιγράφτηκαν προηγουμένως. Για να το πετύχουμε αυτό διατηρούμε σταθερό τον χρόνο εκτέλεσης του αλγορίθμου σε $\Theta(\log n)$, ενώ μειώνουμε τον αριθμό των επεξεργαστών από $\Theta(n)$ σε $\Theta(n/\log n)$.

Θα υλοποιηθεί ο βέλτιστος αλγόριθμος για τον πρώτο αλγόριθμο που αναλύσαμε, δηλαδή για τον αλγόριθμο που αναπτύχθηκε στο μοντέλο EREW PRAM. Ο βέλτιστος αλγόριθμος χωρίζεται σε τρείς φάσεις.

Στη πρώτη φάση, ο καθένας από τους $n/\log n$ επεξεργαστές που χρησιμοποιούμε υπολογίζει σειριακά το προθεματικό άθροισμα για μια υπολίστα από $\log n$ στοιχεία από τα n στοιχεία της κοινόχρηστης μνήμης. Οι σειριακοί υπολογισμοί του προθεματικού αθροίσματος γίνονται παράλληλα για όλους τους επεξεργαστές. Ο σειριακός αλγόριθμος που χρησιμοποιείται είναι ο αλγόριθμος που περιγράφτηκε στην αρχή του κεφαλαίου αυτού, αλλά εκτελείται για $\log n$ δεδομένα εισόδου κάθε φορά αντί για n.

Έπειτα, στη δεύτερη φάση εκτελείται ο πρώτος παράλληλος αλγόριθμος προθεματικού υπολογισμού με δεδομένα εισόδου από τα αποτελέσματος της πρώτης φάσης. Συγκεκριμένα, οι $n/\log n$ επεξεργαστές τρέχουν τον αλγόριθμο μόνο για το τελευταίο στοιχείο της κάθε υπολίστας με τα $\log n$ στοιχεία που χρησιμοποιήθηκαν στο προηγούμενο μέρος. Το σύνολο των δεδομένων εισόδου είναι $n/\log n$, όσες ήταν και οι υπολίστες στις οποίες εκτελέστηκε ο σειριακός αλγόριθμος στη προηγούμενη φάση. Συγκεκριμένα, τα δεδομένα εισόδου είναι τα στοιχεία στη θέση $i \log n$ της κοινόχρηστης μνήμης, όπου $i = 1, 2, .., n/\log n$. Μετά το τέλος της εκτέλεσης του αυτής της φάσης παίρνουμε ως αποτέλεσμα $n/\log n$ προθέματα.

Στην τρίτη και τελευταία φάση του αλγορίθμου αυτού, χρησιμοποιούνται $n/\log n - 1$ επεξεργαστές. Τα δεδομένα χωρίζονται και εδώ σε υπολίστες ίδιες με αυτές της πρώτης φάσης, οι οποίες όμως δεν περιλαμβάνουν το τελευταίο τους στοιχείο. Ο κάθε επεξεργαστής είναι υπεύθυνος για να προβεί σε υπολογισμούς για τα $\log n - 1$ στοιχεία μιας από τις υπολίστες. Συγκεκριμένα, οι επεξεργαστές προσθέτουν σε κάθε ένα από τα $\log n - 1$ στοιχεία ένα από τα προθέματα που υπολογίστηκε στη φάση δύο. Ο κάθε επεξεργαστής προσθέτει το ίδιο πρόθεμα στα στοιχεία που έχει στη διάθεσή του, αλλά το πρόθεμα αυτό είναι διαφορετικό από τα προθέματα των υπόλοιπων επεξεργαστών. Οι επεξεργαστές προσθέτουν στα στοιχεία μιας υπολίστας το πρόθεμα που υπολογίστηκε προηγουμένως και άνηκε στην προηγούμενη υπολίστα. Δηλαδή, ο επεξεργαστής P_i προσθέτει το ίδιο πρόθεμα $\pi_{(i-1)\log n}$ στα στοιχεία της υπολίστας L_i.

Ο αλγόριθμος για το βέλτιστο προθεματικό άθροισμα στο μοντέλο EREW είναι ο ακόλουθος [5]:

Φάση 1:

$Processors\ i = 1\ to\ n/\log n\ \ do\ in\ parallel$

$S_{[(i-1)\log n]+1} = X_{[(i-1)\log n]+1}$

$for\ j = 2\ to\ \log n\ \ do$

$S_{[(i-1)\log n]+j} = S_{[(i-1)\log n]+j-1} + X_{[(i-1)\log n]+j}$

$end\ do$

$end\ do$

Φάση 2:

$Processors\ i = 1\ to\ n/\log n\ \ Run\ Algorithm\ 1$

$On(\ S_{\log n}, S_{2\log n},\ S_{3\log n}, \dots, S_{[\mathrm{n}/\log n]\log n}\ )$

Φάση 3:

$Processors\ i = 2\ to\ n/\log n\ \ do\ in\ parallel$

$for\ j = 1\ to\ \log n - 1\ do$

$S_{[(i-1)\log n]+j} = S_{[(i-1)\log n]} + S_{[(i-1)\log n]+j}$

Η πρώτη φάση του αλγορίθμου χρειάζεται χρόνο $\Theta(\log n)$ για να εκτελεστεί, αφού υπολογίζει σειριακά τα προθέματα για $\log n$ στοιχεία. Εξάλλου, ο σειριακός αλγόριθμος προθεματικού αθροίσματος όπως εξηγήσαμε και στην αρχή του κεφαλαίου αυτού εκτελείται σε χρόνο $\Theta(n)$ για n στοιχεία εισόδου. Άρα, αφού εκτελούμε τον ίδιο αλγόριθμο αλλά με $\log n$ στοιχεία εισόδου ο χρόνος εκτέλεσης της φάσης αυτής είναι λογικό να είναι $\Theta(\log n)$.

Στη δεύτερη φάση του βέλτιστου αλγορίθμου όπου τρέχουμε τον πρώτο παράλληλο αλγόριθμο για προθεματικό άθροισμα, χρειαζόμαστε χρόνο $\Theta(\log n)$ για την εκτέλεση του. Όπως είδαμε και στο υποκεφάλαιο που αναλύθηκε ο αλγόριθμος αυτός, χρειάζεται χρόνος $\Theta(\log n)$ για την εκτέλεση του με n στοιχεία εισόδου. Στην περίπτωση μας, που τα στοιχεία εισόδου είναι $n/\log n$ ο χρόνος που απαιτείται είναι $\Theta(\log(n/\log n))$. Βάση των ιδιοτήτων των λογαρίθμων αυτό σημαίνει ότι ο χρόνος ισούται με $\Theta(\log n - \log(\log n))$. Ο λογάριθμος $\log(\log n)$ δίνει μια μικρή τιμή, κατά πολύ μικρότερη από την τιμή που δίνει ο λογάριθμος $\log n$. Άρα, μπορούμε να πούμε ότι ισχύει $\log(\log n)$ είναι της τάξης του $\log n$. Ως αποτέλεσμα, ο χρόνος εκτέλεσης του αλγορίθμου της δεύτερης φάσης είναι της τάξης $\Theta(\log n)$.

Όπως και οι δύο προηγούμενες φάσεις, έτσι και αυτή χρειάζεται χρόνο $\Theta(\log n)$ για να εκτελεστεί. Στη φάση αυτή γίνεται το άθροισμα ενός προθέματος σε $\log n - 1$ στοιχεία μιας υπολίστας. Άρα, υπάρχει μια επανάληψη η οποία εκτελείται $\log n - 1$ φορές. Στο εσωτερικό της επανάληψης απλά διαβάζονται στοιχεία από την κοινόχρηστη μνήμη και υπολογίζεται ένα άθροισμα. Οι πράξεις αυτές εκτελούνται σε σταθερό χρόνο, $\Theta(1)$. Επομένως, ο χρόνος που απαιτείται για την εκτέλεσης της τρίτης φάσης είναι $\Theta(\log n)$.
Αφού και στις τρεις φάσεις του βέλτιστου αλγορίθμου χρειάζεται χρόνος $\Theta(\log n)$ για την ορθή εκτέλεσή τους, ο συνολικός χρόνος εκτέλεσης του είναι $\Theta(\log n) + \Theta(\log n) + \Theta(\log n) = \Theta(\log n)$ [5].

Στο σημείο αυτό θα υπολογίσουμε το κόστος του βέλτιστου παράλληλου αλγορίθμου προθεματικού αθροίσματος. Όπως αναφέραμε προηγουμένως, σε κάθε μια από τις δύο πρώτες φάσεις του αλγορίθμου χρησιμοποιούνται $n/\log n$ επεξεργαστές. Αφού ο χρόνος εκτέλεσης είναι και στις δύο περιπτώσεις $\Theta(\log n)$, τότε το κόστος τους είναι $(n/\log n) * \Theta(\log n) = \Theta(n)$. Το ίδιο ισχύει και για την τρίτη φάση όπου

χρησιμοποιούνται $n/\log n - 1$ επεξεργαστές, δηλαδή επεξεργαστές της τάξης $\Theta(n/\log n)$. Άρα, και στη τρίτη φάση απαιτείται κόστος $\Theta(n)$. Έτσι, η συνολική πολυπλοκότητα κόστους του αλγορίθμου αυτού για προθεματικό άθροισμα είναι της τάξης $\Theta(n)$, όπως ήταν και ο αρχικός μας στόχος ώστε ο αλγόριθμος να θεωρείται βέλτιστος [5].

Μέχρι τώρα σε αυτή την ενότητα, μελετήσαμε τρεις διαφορετικούς παράλληλους αλγορίθμους που μπορούν να επιλύσουν το πρόβλημα του προθεματικού αθροίσματος. Και οι τρεις αλγόριθμοι, ο ένας εκ των οποίων είναι βέλτιστος υλοποιούνται με ίδιο θεωρητικό χρόνο εκτέλεσης $\Theta(\log n)$. Όσο αφορά το κόστος, οι δύο πρώτοι αλγόριθμοι έχουν την ίδια πολυπλοκότητα, ενώ ο τρίτος που περιγράψαμε είναι βέλτιστος άρα το κόστος είναι μικρότερο.

Είναι ενδιαφέρων να δούμε αν κάτι τέτοιο ισχύει και στη πραγματικότητα, αφού αν και ασυμπτωτικά έχουν τον ίδιο χρόνο, ο βέλτιστος αλγόριθμος είναι πιο πολύπλοκος από τους άλλους και ιδιαίτερα από τον πρώτο αλγόριθμο.

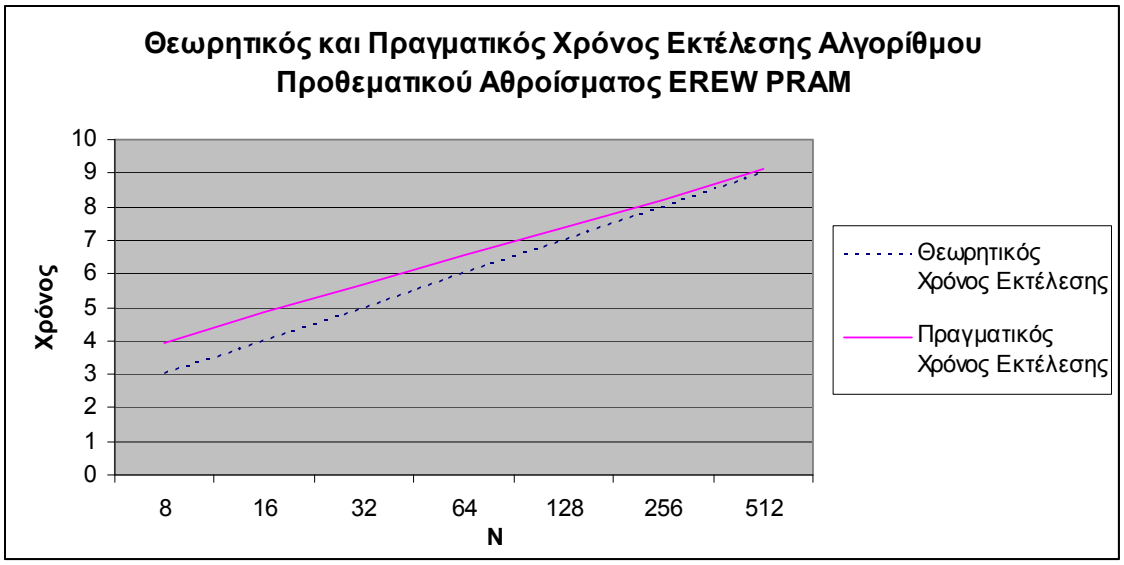
6.6 Πειραματική Αξιολόγηση Μη Βέλτιστων Αλγορίθμων

Αλγόριθμος Προθεματικού Αθροίσματος EREW PRAM

Ο παράλληλος αλγόριθμος προθεματικού αθροίσματος που υλοποιήθηκε στο μοντέλο EREW PRAM, αναπτύχθηκε με την χρήση της γλώσσας XMTC και προσομοιώθηκε στον προσομοιωτή XMT για να πάρουμε τα αποτελέσματα. Πιο κάτω φαίνονται ο μετρήσεις που πάρθηκαν από την προσομοίωση, για αξιολόγηση του παράλληλου αλγορίθμου του προθεματικού αθροίσματος στο μοντέλο EREW PRAM. Οι μετρήσεις πάρθηκαν από την εκτέλεση του αλγορίθμου με διαφορετικό αριθμό δεδομένων εισόδου n κάθε φορά. Στον ακόλουθο πίνακα μπορούμε να δούμε τον χρόνο (msec) και το κόστος εκτέλεσης του αλγορίθμου με n δεδομένα εισόδου και με τη χρήση P επεξεργαστών. Επίσης, στον πίνακα φαίνεται ο χρόνος $(\log n)$ που θεωρητικά πρέπει να έχει ο αλγόριθμος βάση της ασυμπτωτικής ανάλυσης.

n	P	$T = \log n$	Χρόνος	Κόστος
8	7	3	3.96	27.72
16	15	4	4.86	72.90
32	31	5	5.70	176.70
64	63	6	6.55	412.65
128	127	7	7.37	935.99
256	255	8	8.21	2093.55
512	511	9	9.10	4650.10

Πίνακας 6.1 – Μετρήσεις Προσομοίωσης



Γραφική 6.1 – Σχέση Θεωρητικού και Πραγματικού Χρόνου Εκτέλεσης

Από τις μετρήσεις συμπεραίνουμε ότι ο χρόνος εκτέλεσης του αλγορίθμου εξαρτάται από τον αριθμό των δεδομένων εισόδου. Συγκεκριμένα, όπως παρατηρούμε από την πιο πάνω γραφική παράσταση, όταν ο αριθμός των δεδομένων αυξάνεται τότε αυξάνεται λογαριθμικά και ο χρόνος που χρειάζεται για την ολοκλήρωση του αλγόριθμου. Το κόστος υπολογίζεται βάση του αριθμού των δεδομένων εισόδου και του χρόνου εκτέλεσης. Άρα, και το κόστος εξαρτάται παρομοίως από το πλήθος των δεδομένων εισόδου. Αυτό είναι αναμενόμενο, αφού τόσο το κόστος όσο και ο χρόνος εκτέλεσης ασυμπτωτικά δηλώνεται βάσει του πλήθους των δεδομένων εισόδου.

Είναι ενδιαφέρον να προσέξουμε τη σχέση μεταξύ του θεωρητικού χρόνου εκτέλεσης του αλγορίθμου και του ουσιαστικού χρόνου που χρειάζεται για να ολοκληρωθεί η εκτέλεση του αλγορίθμου. Απ' ότι παρατηρούμε, τα δύο στοιχεία του πίνακα που αφορούν το χρόνο έχουν παρόμοιες τιμές. Ο πραγματικός χρόνος που χρειάστηκε για την εκτέλεση του αλγορίθμου είναι πολύ κοντά στον θεωρητικό χρόνο, αλλά δεν είναι ο ίδιος. Αυτό είναι λογικό γιατί ο θεωρητικός χρόνος είναι ασυμπτωτικός, δηλαδή δεν λαμβάνει υπόψη του ένα σύνολο από σταθερές που αυξάνουν τον πραγματικό χρόνο εκτέλεσης. Μπορούμε επίσης να παρατηρήσουμε ότι όσο αυξάνονται τα δεδομένα εισόδου οι σταθερές οι οποίες αυξάνουν τον πραγματικό χρόνο εκτέλεσης εξαλείφονται, και έτσι ο χρόνος που πάρθηκε από τις μετρήσεις πλησιάζει ακόμα πιο πολύ τον θεωρητικό χρόνο εκτέλεσης.

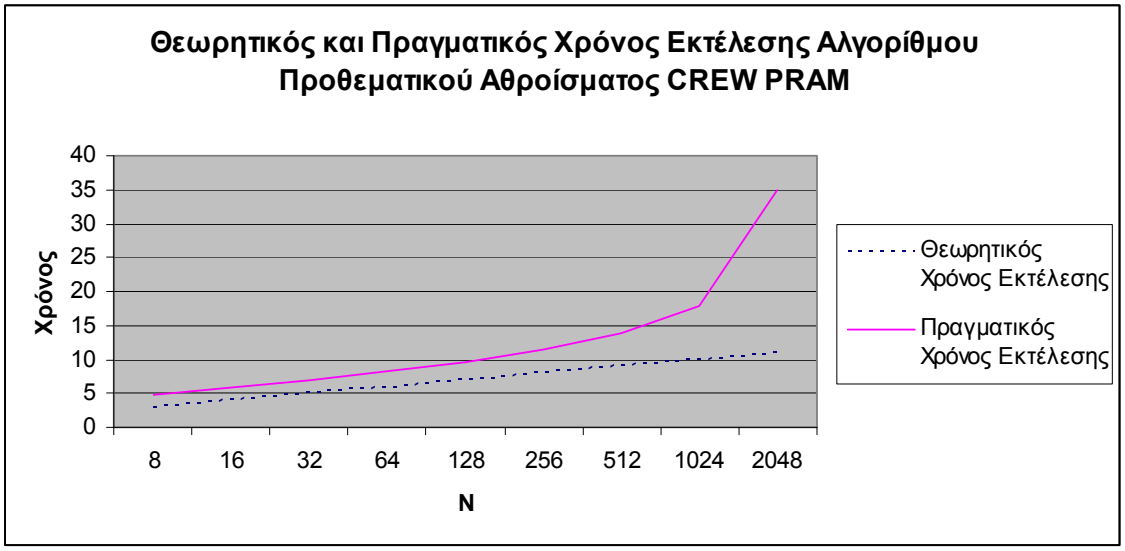
Αφού το κόστος εξαρτάται από τον χρόνο εκτέλεσης του αλγορίθμου, όσα παρατηρήθηκαν στην προηγούμενη παράγραφο για τη σχέση μεταξύ θεωρητικού και πραγματικού χρόνου εκτέλεσης του αλγορίθμου ισχύουν και για το κόστος.

Αλγόριθμος Προθεματικού Αθροίσματος CREW PRAM

Ο παράλληλος αλγόριθμος προθεματικού αθροίσματος που υλοποιήθηκε στο μοντέλο CREW PRAM, αναπτύχθηκε με την χρήση της γλώσσας XMTC και προσομοιώθηκε στον προσομοιωτή XMT για να πάρουμε τα αποτελέσματα. Πιο κάτω φαίνονται ο μετρήσεις που πάρθηκαν από την προσομοίωση, για αξιολόγηση του παράλληλου αλγορίθμου του προθεματικού αθροίσματος στο μοντέλο CREW PRAM. Οι μετρήσεις πάρθηκαν από την εκτέλεση του αλγορίθμου με διαφορετικό αριθμό δεδομένων εισόδου n κάθε φορά. Στον ακόλουθο πίνακα μπορούμε να δούμε τον χρόνο (msec) και το κόστος εκτέλεσης του αλγορίθμου με n δεδομένα εισόδου και με τη χρήση P επεξεργαστών. Επίσης, στον πίνακα φαίνεται ο χρόνος ($\log n$) που θεωρητικά πρέπει να έχει ο αλγόριθμος βάση της ασυμπτωτικής ανάλυσης.

n	P	$T = \log n$	Χρόνος	Κόστος
8	4	3	4.75	19.00
16	8	4	5.85	46.80
32	16	5	6.95	111.20
64	32	6	8.15	260.80
128	64	7	9.63	616.32
256	128	8	11.41	1460.48
512	256	9	13.84	3543.04
1024	512	10	17.95	9190.40
2048	1024	11	34.92	35758.08

Πίνακας 6.2 – Μετρήσεις Προσομοίωσης



Γραφική 6.2 – Σχέση Θεωρητικού και Πραγματικού Χρόνου Εκτέλεσης

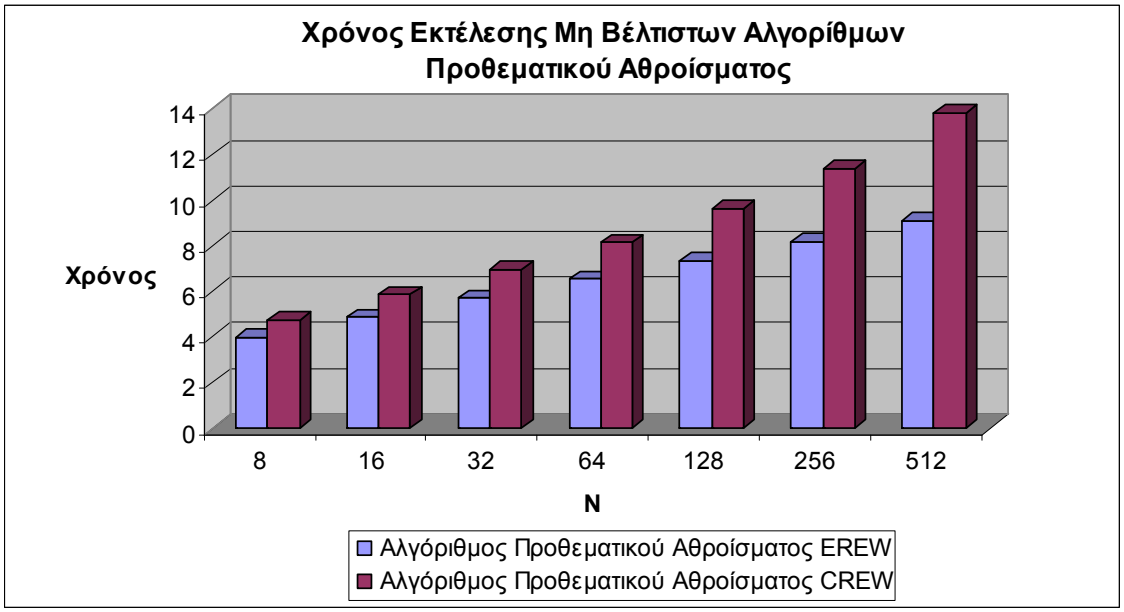
Από τις πιο πάνω μετρήσεις παρατηρούμε ότι όπως και στην περίπτωση του αλγορίθμου προθεματικού αθροίσματος στο μοντέλο EREW PRAM, ο χρόνος και το κόστος εξαρτούνται από τον αριθμό των δεδομένων εισόδου. Συγκεκριμένα, όταν το πλήθος των δεδομένων εισόδου αυξάνεται, τότε αυξάνεται λογαριθμικά και ο χρόνος που χρειάστηκε για την εκτέλεση του αλγορίθμου. Το γεγονός αυτό επιβεβαιώνει την θεωρητική ανάλυση, σύμφωνα με την οποία ο χρόνος εκτέλεσης του αλγορίθμου είναι λογαριθμικής τάξης. Επιπρόσθετα, αφού το κόστος υπολογίζεται βάσει του αριθμού

των δεδομένων εισόδου και του χρόνου εκτέλεσης εξαρτάται και αυτό από το πλήθος των δεδομένων εισόδου. Αυτό είναι αναμενόμενο, αφού τόσο το κόστος όσο και ο χρόνος εκτέλεσης ασυμπτωτικά δηλώνεται βάσει του πλήθους των δεδομένων εισόδου.

Συγκρίνοντας τον θεωρητικό χρόνο εκτέλεσης και τον πραγματικό χρόνο που χρειάστηκε για να ολοκληρωθεί η εκτέλεση του αλγορίθμου, παρατηρούμε ότι έχουν κάποια διαφορά. Η διαφορά αυτή μεταξύ των δύο χρόνων αυξάνεται σταθερά καθώς αυξάνεται και ο αριθμός των δεδομένων εισόδου. Όσο πιο μεγάλος είναι ο αριθμός των δεδομένων τόσο μεγαλύτερη είναι η απόκλιση του πραγματικού χρόνου εκτέλεσης από τον θεωρητικό χρόνο, σε αντίθεση με την περίπτωση του αλγορίθμου στο μοντέλο EREW PRAM. Το γεγονός αυτό είναι δικαιολογημένο αφού όπως αναφέρθηκε και στο κεφάλαιο που περιγράφτηκε η γλώσσα XMTC, στην περίπτωση που υπάρχει ταυτόχρονη ανάγνωση την ίδιας κυψελίδας μνήμης από περισσότερους από ένα επεξεργαστές, χρειάζεται περισσότερος χρόνος για την πρόσβαση στη θέση μνήμης σε σχέση με τον χρόνο που χρειάζεται όταν ένας μόνο επεξεργαστής έχει πρόσβαση στη θέση αυτή. Για αυτό, όταν ο αριθμός των δεδομένων είναι μεγαλύτερος, δηλαδή όταν υπάρχουν περισσότερες προσβάσεις στην μνήμη ο συνολικός χρόνος εκτέλεσης του αλγορίθμου αυξάνεται και υπάρχει μεγαλύτερη απόκλιση από τον θεωρητικό χρόνο. Ενώ στην περίπτωση του μοντέλου EREW PRAM κάτι τέτοιο δεν συμβαίνει αφού δεν υπάρχει ταυτόχρονη πρόσβαση στην ίδια κυψελίδα της κοινόχρηστης μνήμης. Επιπλέον, κάποιος άλλος λόγος της διαφοράς που παρουσιάζεται μεταξύ των δύο χρόνων είναι και ότι στον αλγόριθμο αυτό υπάρχουν περισσότερες πράξεις που ασυμπτωτικά εκτελούνται σε σταθερό χρόνο. Στην πραγματικότητα όμως οι πράξεις αυτές καταναλώνουν κάποιο επιπλέον χρόνο κατά την εκτέλεση. Άρα, όπως και στον προηγούμενο αλγόριθμο η διαφορά μεταξύ των δύο χρόνων εκτέλεσης είναι δικαιολογημένη γιατί ο θεωρητικός χρόνος είναι ασυμπτωτικός και δεν λαμβάνει υπόψη του κάποιες παραμέτρους που αυξάνουν τον πραγματικό χρόνο εκτέλεσης.

Αφού το κόστος εξαρτάται από τον χρόνο εκτέλεσης του αλγορίθμου, όσα παρατηρήθηκαν στην προηγούμενη παράγραφο για τη σχέση μεταξύ θεωρητικού και πραγματικού χρόνου εκτέλεσης του αλγορίθμου ισχύουν και για το κόστος.

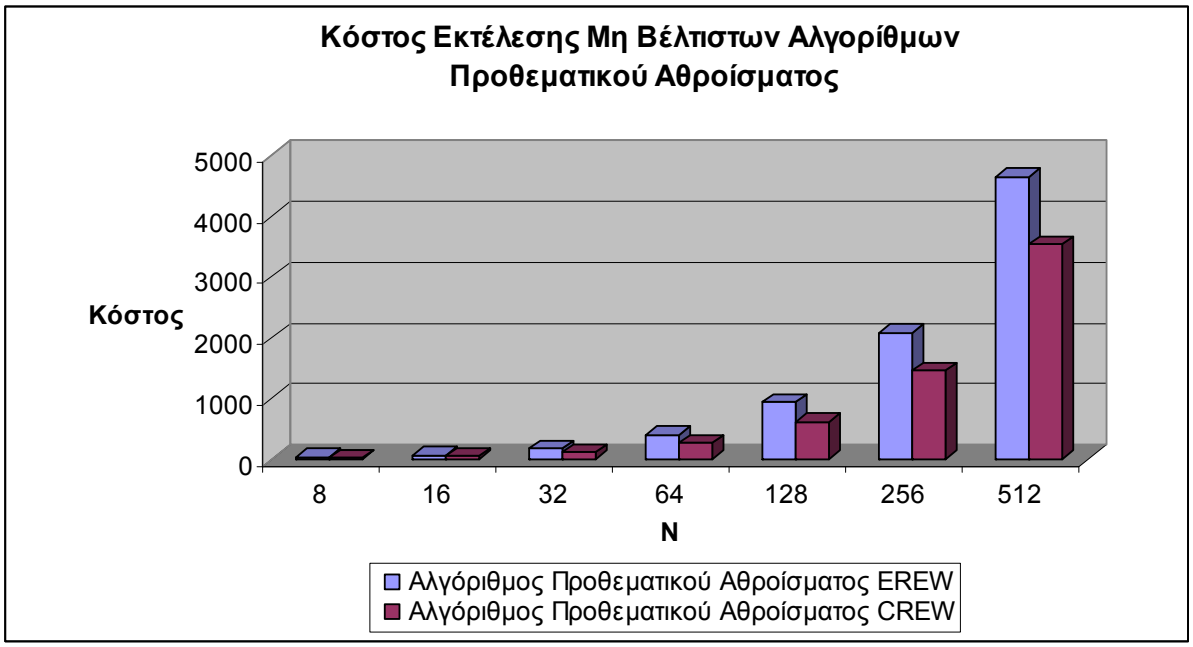
6.7 Σύγκριση Μη Βέλτιστων Αλγορίθμων



Γραφική 6.3 – Σύγκριση Μη Βέλτιστων Αλγορίθμων Προθεματικού Αθροίσματος

Συγκρίνοντας τους δύο αλγορίθμους παρατηρούμε ότι αν και ο θεωρητικός τους χρόνος είναι ο ίδιος, $\Theta(\log n)$, και στις δύο περιπτώσεις, ο πραγματικός τους χρόνος διαφέρει. Συγκεκριμένα ο αλγόριθμος που είναι υλοποιημένος στο μοντέλο EREW PRAM είναι πιο γρήγορος από τον αλγόριθμο που είναι υλοποιημένος στο μοντέλο CREW PRAM. Το γεγονός αυτό συμβαίνει γιατί ο αλγόριθμος που υλοποιείται στο μοντέλο CREW PRAM εκτελεί περισσότερες πράξεις από ότι ο άλλος αλγόριθμος. Αν και ασυμπτωτικά ο χρόνος που χρειάζεται για την εκτέλεση όλων των πράξεων μαζί είναι σταθερός, ουσιαστικά ο μεγαλύτερος αριθμός πράξεων απαιτεί περισσότερο χρόνο. Μπορούμε να παρατηρήσουμε επίσης ότι στις περιπτώσεις όπου ο αριθμός των δεδομένων εισόδου είναι μικρός τότε ο πραγματικός χρόνος εκτέλεσης των δύο αλγορίθμων είναι πιο κοντά ο ένας στον άλλο. Ενώ στην περίπτωση που έχουμε μεγάλο αριθμό δεδομένων η διαφορά του χρόνου μεταξύ των δύο αλγορίθμων είναι αυξάνεται ακόμα περισσότερο. Ο λόγος για τον οποίο συμβαίνει αυτό είναι ότι στη γλώσσα XMTC χρειάζεται περισσότερος χρόνος για πρόσβαση σε μια κυψελίδα της κοινόχρηστης μνήμης όταν συμβαίνει ταυτόχρονη ανάγνωση από περισσότερους από ένα επεξεργαστές της ίδιας κυψελίδας μνήμης. Έτσι, είναι λογικό ο αλγόριθμος ο οποίος προβαίνει σε ταυτόχρονες αναγνώσεις από την μνήμη να χρειάζεται περισσότερο χρόνο για να εκτελεστεί. Επίσης,

στην περίπτωση που έχουμε περισσότερα δεδομένα εισόδου, βάσει του αλγορίθμου υπάρχουν περισσότερες προσβάσεις ταυτόχρονα στην ίδια κυψελίδα μνήμης. Για αυτό και στους μεγαλύτερους αριθμούς δεδομένων εισόδου ο χρόνος μεταξύ των δύο αλγορίθμων αποκλίνει ακόμα περισσότερο. Επομένως, είναι απόλυτα λογικό ο αλγόριθμος προθεματικού αθροίσματος στο μοντέλο EREW PRAM να είναι πιο γρήγορος από τον αντίστοιχο αλγόριθμο στο μοντέλο CREW PRAM.



Γραφική 6.4 – Σύγκριση Μη Βέλτιστων Αλγορίθμων Προθεματικού Αθροίσματος

Όσον αφορά το κόστος, παρατηρούμε ότι ο αλγόριθμος που είναι υλοποιημένος στο μοντέλο CREW PRAM είναι καλύτερος απ' αυτόν που είναι υλοποιημένος στο EREW PRAM, παρόλο που θεωρητικά οι δύο αλγόριθμοι έχουν ίδιο κόστος. Επιπλέον, αυτό έρχεται και σε αντίφαση με το γεγονός ότι ο πραγματικός χρόνος εκτέλεσης του αλγορίθμου στο μοντέλο EREW PRAM είναι μικρότερος και άρα θα έπρεπε και το κόστος του να είναι μικρότερο.

Το αποτέλεσμα αυτό οφείλεται στο γεγονός ότι το θεωρητικό κόστος υπολογίζεται ασυμπτωτικά και για αυτό είναι κοινό και στους δύο αλγόριθμους. Ουσιαστικά όμως, υπάρχει διαφορά στον αριθμό των επεξεργαστών που χρησιμοποιεί ο κάθε αλγόριθμος. Ο αλγόριθμος προθεματικού αθροίσματος στο μοντέλο EREW PRAM χρησιμοποιεί για την εκτέλεση των εντολών $n-1$ επεξεργαστές, ενώ ο άλλος αλγόριθμος χρησιμοποιεί

$n/2$ επεξεγραστές. Αν και ασυμπτωτικά ο αριθμός των επεξεργαστών που χρησιμοποιούν οι αλγόριθμοι είναι και στις δύο περιπτώσεις $\Theta(n)$, στην πραγματικότητα ο δεύτερος αλγόριθμος χρησιμοποιεί περίπου τους μισούς επεξεργαστές σε σχέση με τον πρώτο αλγόριθμο.

Έτσι, αφού το κόστος εξαρτάται και από τον χρόνο εκτέλεσης ενός αλγορίθμου και από τον αριθμό των επεξεργαστών που χρησιμοποιεί είναι δικαιολογημένο να έχουμε τα αποτελέσματα αυτά. Δεν μπορούμε ουσιαστικά να πούμε ότι τα αποτελέσματα είναι αντιφατικά αφού η διαφορά μεταξύ του αριθμού των επεξεργαστών είναι μεγαλύτερη από την διαφορά στο χρόνο που παρουσιάζουν οι δύο αλγόριθμοι.

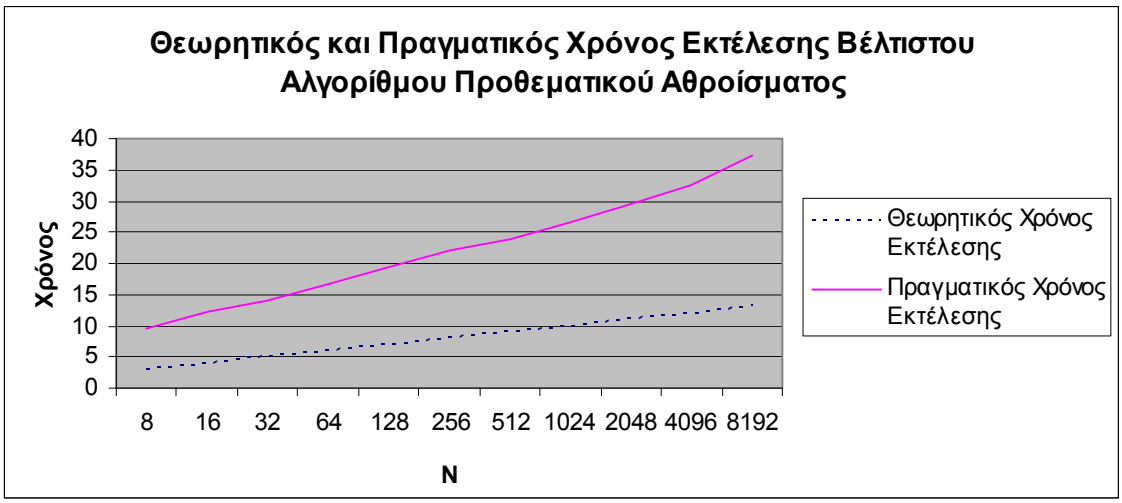
Συνεπώς, από τη σύγκριση των δύο αλγορίθμων καταλήγουμε στο ότι ο πρώτος αλγόριθμος που είναι σχεδιασμένος για το μοντέλο EREW PRAM έχει καλύτερο χρόνο εκτέλεσης, ενώ ο δεύτερος αλγόριθμος που είναι σχεδιασμένος στο CREW PRAM έχει καλύτερο κόστος. Άρα, στην περίπτωση που δεν μας ενδιαφέρει τόσο το κόστος εκτέλεσης ο αλγόριθμος στο EREW PRAM είναι πιο αποδοτικός αφού υπάρχει μεγάλη διαφορά με τον αλγόριθμο CREW PRAM, ιδιαίτερα για μεγαλύτερο πλήθος δεδομένων εισόδου. Στην περίπτωση που το κόστος εκτέλεσης είναι σημαντικό τότε θεωρούμε πιο αποδοτικό τον αλγόριθμο στο μοντέλο CREW PRAM.

6.8 Πειραματική Αξιολόγηση Βέλτιστου Αλγορίθμου

Ο βέλτιστος παράλληλος αλγόριθμος προθεματικού αθροίσματος που υλοποιήθηκε στο μοντέλο EREW PRAM, αναπτύχθηκε με την χρήση της γλώσσας XMTC και προσομοιώθηκε στον προσομοιωτή XMT για να πάρουμε τα αποτελέσματα. Πιο κάτω φαίνονται ο μετρήσεις που πάρθηκαν από την προσομοίωση, για αξιολόγηση του παράλληλου αλγορίθμου του προθεματικού αθροίσματος στο μοντέλο EREW PRAM. Οι μετρήσεις πάρθηκαν από την εκτέλεση του αλγορίθμου με διαφορετικό αριθμό δεδομένων εισόδου n κάθε φορά. Στον ακόλουθο πίνακα μπορούμε να δούμε τον χρόνο (msec) και το κόστος εκτέλεσης του αλγορίθμου με n δεδομένα εισόδου και με τη χρήση P επεξεργαστών. Επίσης, στον πίνακα φαίνεται ο χρόνος ($\log n$) που θεωρητικά πρέπει να έχει ο αλγόριθμος βάση της ασυμπτωτικής ανάλυσης.

n	P	$T = \log n$	Χρόνος	Κόστος
8	3	3	9.41	28.23
16	4	4	12.19	48.76
32	7	5	14.08	98.56
64	11	6	16.62	182.82
128	19	7	19.36	367.84
256	32	8	22.07	706.24
512	57	9	23.92	1363.44
1024	103	10	26.62	2741.86
2048	187	11	29.45	5507.15
4096	342	12	32.61	11152.62
8192	631	13	37.17	23454.27

Πίνακας 6.3 – Μετρήσεις Προσομοίωσης



Γραφική 6.5 – Σχέση Θεωρητικού και Πραγματικού Χρόνου Εκτέλεσης

Από τις πιο πάνω μετρήσεις παρατηρούμε ότι όπως και στις προηγούμενες περιπτώσεις παράλληλων αλγορίθμων προθεματικού αθροίσματος, έτσι και σε αυτόν ο χρόνος και το κόστος εξαρτούνται από τον αριθμό των δεδομένων εισόδου. Συγκεκριμένα, όταν το πλήθος των δεδομένων εισόδου αυξάνεται, τότε αυξάνεται λογαριθμικά ο χρόνος που χρειάστηκε για την εκτέλεση του αλγορίθμου. Αφού το κόστος υπολογίζεται βάση του αριθμού των δεδομένων εισόδου και του χρόνου εκτέλεσης εξαρτάται και αυτό από το πλήθος των δεδομένων εισόδου. Αυτό είναι αναμενόμενο, αφού τόσο το κόστος όσο

και ο χρόνος εκτέλεσης ασυμπτωτικά δηλώνεται βάσει του πλήθους των δεδομένων εισόδου.

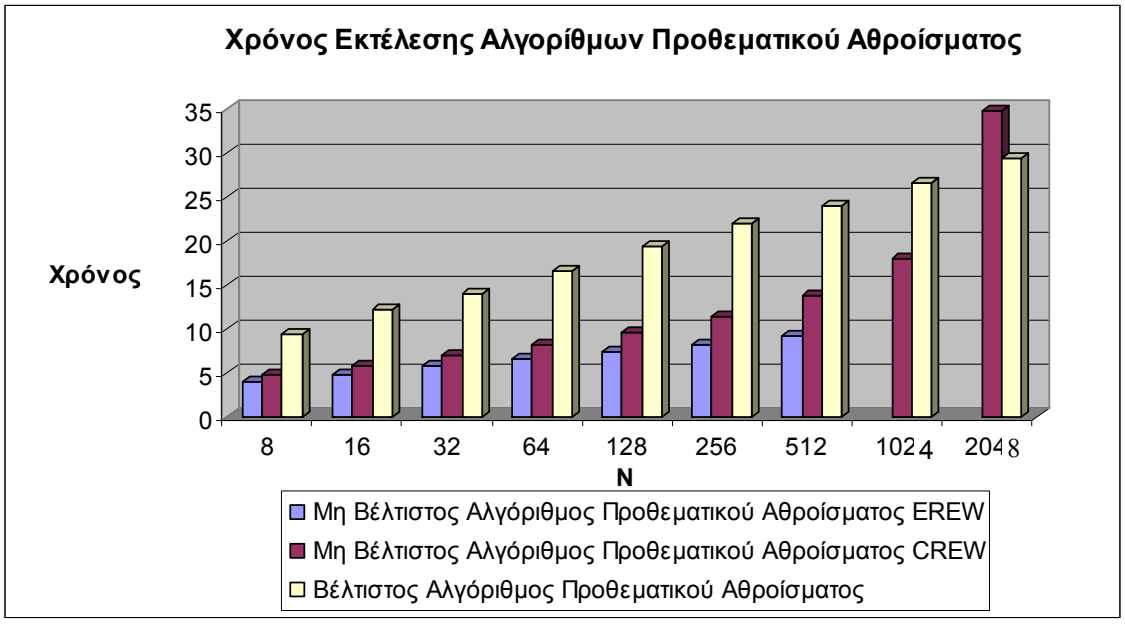
Μεγάλο ενδιαφέρον παρουσιάζει η σχέση μεταξύ του θεωρητικού χρόνου εκτέλεσης και του πραγματικού χρόνου εκτέλεσης του βέλτιστου αλγορίθμου προθεματικού αθροίσματος. Όπως είναι φυσικό ο πραγματικός χρόνος εκτέλεσης είναι μεγαλύτερος από τον θεωρητικό. Ο ρυθμός αύξησης όμως του χρόνου εκτέλεσης είναι λογαριθμικός, άρα ο χρόνος εκτέλεσης έχει τον ίδιο ρυθμό αύξησης με τον θεωρητικό χρόνο. Έτσι, αν και ο πραγματικός χρόνος είναι μεγαλύτερος από τον θεωρητικό, επιβεβαιώνεται η θεωρητική ανάλυση.

Η διαφορά μεταξύ πραγματικού και θεωρητικού χρόνου οφείλεται στην πολυπλοκότητα των πράξεων που εκτελούνται, αφού όπως αναφέραμε και προηγουμένως ο αλγόριθμος αποτελείται από τρεις φάσεις. Η κάθε μία από αυτές τις τρεις φάσεις εκτελείται θεωρητικά σε χρόνο $\Theta(\log n)$. Επομένως, είναι λογικό ο πραγματικός χρόνος να διακυμαίνεται σε τιμές υψηλότερες από το $\log n$, αφού ο αλγόριθμος αποτελείται από τρία μέρη που χρειάζονται χρόνο $\Theta(\log n)$ για να εκτελεστούν.

Τα αντίστοιχα συμπεράσματα που εξάχθηκαν για τον χρόνο ισχύουν και για την σχέση μεταξύ του πραγματικού και θεωρητικού κόστους, αφού το κόστος εξαρτάται τόσο από τον χρόνο, όσο και από το πλήθος των δεδομένων εισόδου του αλγορίθμου. Στην περίπτωση του κόστους όμως ο ρυθμός αύξησης που παρατηρείται είναι γραμμικός.

Τέλος, και σε αυτό τον αλγόριθμο το πλήθος των επεξεργαστών εξαρτάται από το πλήθος των δεδομένων εισόδου. Συγκεκριμένα, όπως προαναφέρθηκε, για n στοιχεία εισόδου χρειάζονται $n/\log n$ επεξεργαστές για την σωστή εκτέλεση του αλγορίθμου. Στην περίπτωση που ο υπολογισμός $n/\log n$ δεν δίνει ακέραιο αριθμό, τότε ο αριθμός των επεξεργαστών που χρησιμοποιούνται είναι ο επόμενος ακέραιος αριθμός από την τιμή που παίρνουμε.

6.9 Σύγκριση Αλγορίθμων



Γραφική 6.6 – Σύγκριση Αλγορίθμων Προθεματικού Αθροίσματος

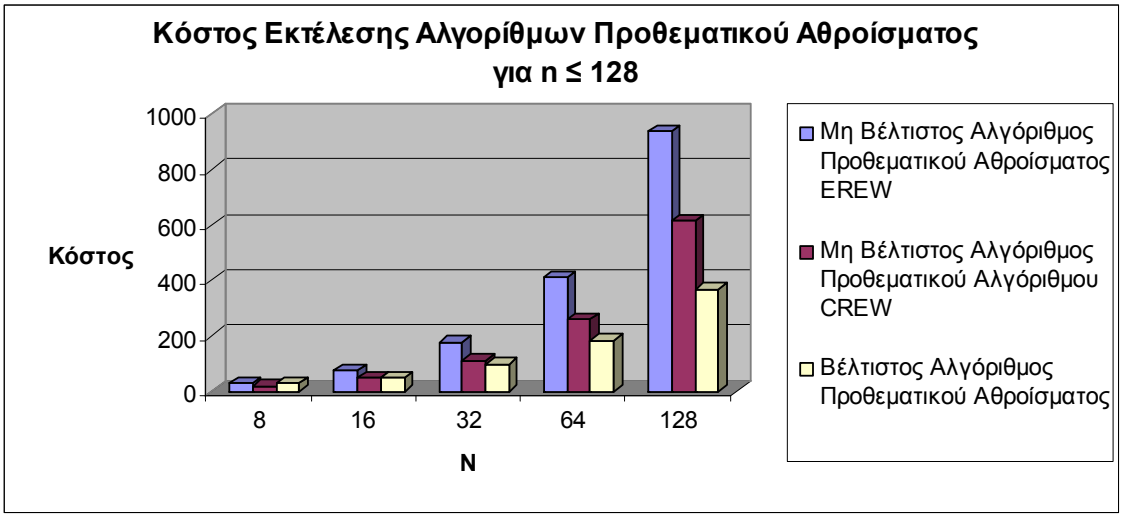
Παρατηρώντας την πιο πάνω γραφική παράσταση που αφορά τον χρόνο εκτέλεσης των τριών αλγορίθμων προθεματικού αθροίσματος που μελετήθηκαν, συμπεραίνουμε ότι υπάρχουν εμφανές διαφορές μεταξύ τους. Σύμφωνα με την θεωρητική αξιολόγηση των αλγορίθμων, έχουν και οι τρεις τον ίδιο ασυμπτωτικό χρόνο $\Theta(\log n)$.

Παρατηρούμε όμως ότι οι δύο μη βέλτιστοι αλγόριθμοι έχουν μεταξύ τους μια μικρή διαφορά στον χρόνο εκτέλεσής, που αυξάνεται με την αύξηση του αριθμού των δεδομένων εισόδου. Το σημαντικότερο όμως είναι η διαφορά που παρουσιάζεται μεταξύ των δύο μη βέλτιστων και του βέλτιστου αλγορίθμου, η οποία είναι αρκετά μεγάλη. Για μεγάλο όμως πλήθος δεδομένων εισόδου, που είναι και οι περιπτώσεις που μας ενδιαφέρουν περισσότερο, ο χρόνος εκτέλεσης πλησιάζει περισσότερο το χρόνο των μη βέλτιστων αλγορίθμων. Για πλήθος δεδομένων n=2048, ο βέλτιστος αλγόριθμος γίνεται ταχύτερος από τον μη βέλτιστο.

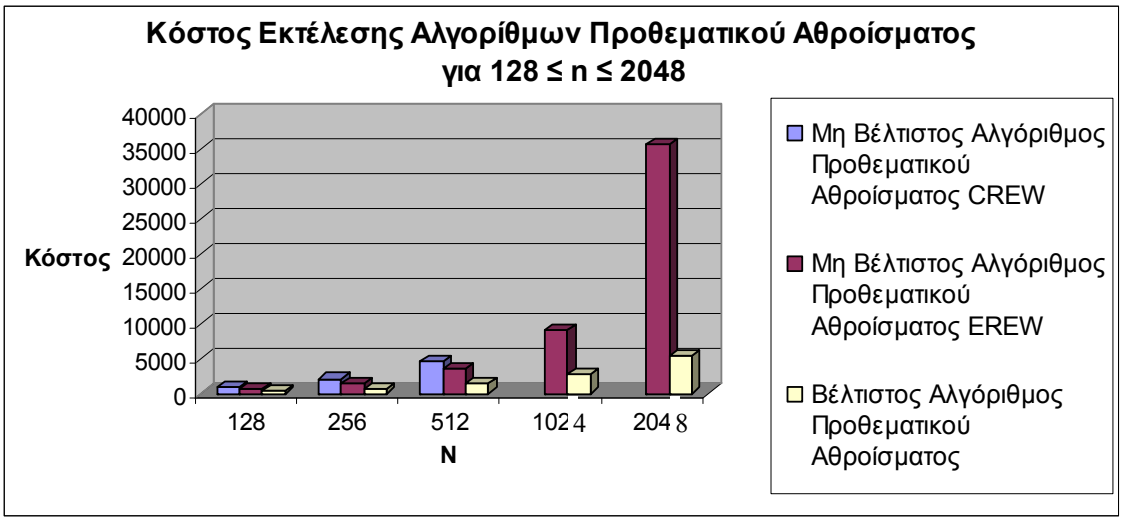
Το γεγονός ότι ο βέλτιστος αλγόριθμος είναι σχεδόν σε όλες τις περιπτώσεις πιο αργός μπορεί να δικαιολογηθεί απόλυτα. Όπως έχουμε αναφέρει και πριν, ο βέλτιστος αλγόριθμος είναι πιο πολύπλοκος αφού χωρίζεται σε τρεις φάσεις. Αρχικά την φάση του σειριακού υπολογισμού προθεμάτων για $\log n$ στοιχεία, έπειτα τον πρώτο

προθεματικό αλγόριθμο που είδαμε να υλοποιείται στο μοντέλο EREW PRAM, και τέλος την φάση αθροίσματος ενός προθέματος με κάποιο πλήθος στοιχείων. Κάθε μια από αυτές τις φάσεις χρειάζεται χρόνο $\Theta(\log n)$ για να εκτελεστεί, χρόνο που απαιτούν ολόκληροι οι άλλοι δύο αλγόριθμοι. Ασυμπτωτικά ο χρόνος είναι ο ίδιος, αλλά είναι φανερό ότι ο βέλτιστος αλγόριθμος αφού αποτελείται από τρεις φάσεις που εκτελούνται σε χρόνο $\Theta(\log n)$, στην πραγματικότητα χρειάζεται περισσότερο χρόνο για να εκτελεστεί. Εξάλλου, μέρος του βέλτιστου αλγόριθμου είναι και ολόκληρος ο αλγόριθμος προθεματικού υπολογισμού στο μοντέλο EREW PRAM.

Επομένως, δεν μπορούσαμε παρά να αναμένουμε μια τέτοια διαφορά μεταξύ του χρόνου εκτέλεσης του βέλτιστου αλγορίθμου με τους άλλους δύο, έστω κι αν θεωρητικά έχουν την ίδια πολυπλοκότητα χρόνου. Θετικό είναι το γεγονός ότι για δεδομένα εισόδου n=2048 ο βέλτιστος αλγόριθμος φτάνει τον χρόνο των δύο άλλων αλγορίθμων. Αυτό επιβεβαιώνει το γεγονός ότι η ασυμπτωτική ανάλυση ισχύει περισσότερο για μεγάλο πλήθος δεδομένων εισόδου.



Γραφική 6.7 – Σύγκριση Αλγορίθμων Προθεματικού Αθροίσματος



Γραφική 6.8 – Σύγκριση Αλγορίθμων Προθεματικού Αθροίσματος

Σύμφωνα με την θεωρητική αξιολόγηση των αλγορίθμων, το κόστος του βέλτιστου αλγόριθμου πρέπει να είναι καλύτερο από το κόστος των άλλων δύο αλγορίθμων. Αφού ο βέλτιστος χρησιμοποιεί $n / \log n$ επεξεργαστές το κόστος του πρέπει θεωρητικά να είναι της τάξης $\Theta(n)$, σε αντίθεση με τους μη βέλτιστους αλγόριθμους που έχουν κόστος $\Theta(n \log n)$.

Πράγματι, από τις γραφικές παραστάσεις παρατηρούμε ότι το κόστος του βέλτιστου αλγορίθμου είναι καλύτερο σε σχέση με τους μη βέλτιστους αλγόριθμους. Μοναδική εξαίρεση αποτελεί η περίπτωση με τον μικρότερο αριθμό δεδομένων εισόδου, όπου και οι δύο μη βέλτιστοι αλγόριθμοι έχουν σε μικρό βαθμό καλύτερο κόστος. Είναι εμφανείς όμως ότι στις υπόλοιπες περιπτώσεις, όσο αυξάνεται το πλήθος των δεδομένων εισόδου, τόσο περισσότερο αυξάνεται η διαφορά κόστους μεταξύ του βέλτιστου αλγόριθμου και των άλλων δύο.

Στους πίνακες που ακολουθούν μπορούμε να παρατηρήσουμε τις διαφορές που αφορούν την πολυπλοκότητα κόστους μεταξύ του βέλτιστου αλγορίθμου και των δύο μη βέλτιστων.

n	Βέλτιστος Αλγόριθμος	Μη Βέλτιστος Αλγόριθμος EREW	Διαφορά Κόστους
16	48.76	72.90	24.14
64	182.82	412.65	229.83
512	1363.44	4650.10	3286.66

Πίνακας 6.4 – Διαφορές Κόστους

Από τον πιο πάνω πίνακα βλέπουμε την μεγάλη διαφορά κόστους μεταξύ του βέλτιστου αλγορίθμου και του μη βέλτιστου αλγορίθμου που είναι υλοποιημένος στο μοντέλο EREW PRAM. Στον πίνακα παρουσιάζονται δύο από τις ακραίες περιπτώσεις δεδομένων εισόδου, μια μεσαία περίπτωση και η διαφορά κόστους που παρουσιάζεται σε κάθε περίπτωση. Παρατηρούμε ότι καθώς αυξάνεται ο αριθμός των δεδομένων αυξάνεται και η διαφορά κόστους.

n	Βέλτιστος Αλγόριθμος	Μη Βέλτιστος Αλγόριθμος CREW	Διαφορά Κόστους
32	98.56	111.2	12.64
128	367.84	616.32	248.48
2048	5507.15	35758.08	30250.93

Πίνακας 6.5 – Διαφορές Κόστους

Από τον πιο πάνω πίνακα βλέπουμε τη διαφορά κόστους μεταξύ του βέλτιστου αλγορίθμου και του μη βέλτιστου αλγορίθμου που είναι υλοποιημένος στο μοντέλο CREW PRAM. Στον πίνακα παρουσιάζονται δύο από τις ακραίες περιπτώσεις δεδομένων εισόδου, μια μεσαία περίπτωση και η διαφορά κόστους που παρουσιάζεται σε κάθε περίπτωση. Σε αυτή την περίπτωση αν και αρχικά η διαφορά κόστους δεν είναι μεγάλη, στη συνέχεια στις μεσαίες και μεγάλες περιπτώσεις δεδομένων εισόδου η διαφορά αυξάνεται αρκετά. Αποκορύφωμα αποτελεί η διαφορά κόστους που

παρουσιάζεται για το μεγαλύτερο αριθμό δεδομένων εισόδου, η οποία είναι αρκετά μεγαλύτερη από τη διαφορά κόστους που παρουσιάζεται στην αμέσως προηγούμενη περίπτωση.

Επομένως, από την πιο πάνω μελέτη που αφορά το κόστος των αλγορίθμων καταλήξαμε στο ότι το κόστος του βέλτιστου αλγόριθμου είναι σε μεγάλο βαθμό καλύτερο από το κόστος των μη βέλτιστων αλγορίθμων. Αυτό οφείλεται στον μικρό αριθμό των επεξεργαστών που χρησιμοποιούνται για τον βέλτιστο αλγόριθμο. Επιπλέον, σημαντικό είναι ότι το κόστος του βέλτιστου αλγόριθμου βελτιώνεται συνεχώς καθώς αυξάνονται τα δεδομένα εισόδου και με μεγάλη διαφορά από τους δύο άλλους αλγόριθμους. Αυτό συμβαίνει αφού αυξάνοντας το πλήθος των δεδομένων εισόδου η πειραματική αξιολόγηση προσεγγίζει την ασυμπτωτική ανάλυση.

Άρα, επιτύχαμε την υλοποίηση ενός πολύ εξυπηρετικού και αποδοτικού αλγόριθμου όσον αφορά το κόστος για πραγματικά προβλήματα που συνήθως έχουν μεγάλο αριθμό δεδομένων εισόδου. Όμως έχουμε δει ότι όσο αυξάνεται ο αριθμός των δεδομένων εισόδου του βέλτιστου αλγόριθμου τόσο πιο πολύ αυξάνεται ο χρόνος εκτέλεσης του αλγορίθμου και ότι οι μη βέλτιστοι αλγόριθμοι είναι πιο αποδοτικοί σε θέματα χρόνου εκτέλεσης. Επομένως, δεν μπορούμε εύκολα να αποφανθούμε ποιος είναι ο καλύτερος και πιο αποδοτικός αλγόριθμός από τους τρεις. Τα συμπεράσματα στα οποία καταλήγουμε είναι αντιφατικά αφού ο βέλτιστος αλγόριθμος είναι σε μεγάλο βαθμό αποδοτικός σε θέμα κόστους, αλλά απαιτεί μεγάλο χρονικό διάστημα για να εκτελεστεί.

Για να αποφανθούμε αν τελικά ο βέλτιστος αλγόριθμος είναι όντως ο πιο αποδοτικός, όπως στοχεύαμε κατά την υλοποίηση του, πρέπει να μελετήσουμε βαθύτερα τον χρόνο εκτέλεση του αλγορίθμου. Στην περίπτωση που καταλήξουμε σε ένα οριστικό συμπέρασμα που αφορά τον χρόνο, τότε θα μπορέσουμε να πούμε ποιος αλγόριθμος είναι καλύτερος.

Στους πιο κάτω πίνακες παρουσιάζονται οι διαφορές που αφορούν τον χρόνο εκτέλεσης των τριών αλγορίθμων και την διαφορά χρόνου μεταξύ βέλτιστου και μη βέλτιστων αλγορίθμων.

n	**Βέλτιστος Αλγόριθμος**	**Μη Βέλτιστος Αλγόριθμος EREW**	**Διαφορά Χρόνου**
8	9.41	3.96	5.45
64	16.62	6.55	10.07
512	23.92	9.10	14.82

Πίνακας 6.6 – Διαφορές Χρόνου

Στον πιο πάνω πίνακα παρατηρούμε τη διαφορά χρόνου που παρουσιάζεται για σημαντικότερες περιπτώσεις αριθμού δεδομένων εισόδου μεταξύ βέλτιστου και μη βέλτιστου αλγόριθμου. Όπως συμπεραίναμε και προηγουμένως, ο μη βέλτιστος αλγόριθμος είναι πιο γρήγορος από τον βέλτιστο και η διαφορά χρόνου μεταξύ των δύο αλγορίθμων αυξάνεται καθώς το πλήθος των δεδομένων αυξάνεται. Παρόλα αυτά, η διαφορά του χρόνου εκτέλεσης μεταξύ των αλγορίθμων δεν είναι τόσο μεγάλη, ούτε και η διαφορά αυτή αυξάνεται δραματικά από κάποιο αριθμό δεδομένων σε άλλο.

n	**Βέλτιστος Αλγόριθμος**	**Μη Βέλτιστος Αλγόριθμος CREW**	**Διαφορά Χρόνου**
8	9.41	4.75	4.66
64	16.62	8.15	8.47
512	23.92	13.84	10.08

Πίνακας 6.7 – Διαφορές Χρόνου

Στον πιο πάνω πίνακα παρατηρούμε τη διαφορά χρόνου που παρουσιάζεται για σημαντικότερες περιπτώσεις αριθμού δεδομένων εισόδου μεταξύ βέλτιστου και μη βέλτιστου αλγόριθμου. Από τα δεδομένα του πίνακα μπορούμε να καταλήξουμε στα ίδια συμπεράσματα με την περίπτωση του αλγόριθμου του μοντέλου EREW PRAM. Σε αυτό τον αλγόριθμο όμως η διαφορά χρόνου εκτέλεσης μεταξύ των δύο αλγορίθμων

είναι μικρότερη, αφού ο μη βέλτιστος αλγόριθμος στο CREW PRAM δεν είναι τόσο γρήγορος όσο ο αλγόριθμος στο EREW PRAM.

Με βάση όσα μελετήθηκαν στους πιο πάνω πίνακες παρατηρούμε ότι υπάρχει και στην περίπτωση του χρόνου εκτέλεσης διαφορά μεταξύ των αλγορίθμων, αλλά όχι τόσο μεγάλη σε σύγκριση με τη διαφορά κόστους που υπήρχε στους αλγορίθμους. Επιπλέον, αν και υπάρχει αύξηση στη διαφορά χρόνου ανάμεσα στα διαφορετικά πλήθη δεδομένων εισόδου, η αύξηση αυτή δεν φτάνει το μέγεθος της διαφοράς του κόστους.

Επομένως, ο βέλτιστος αλγόριθμος είναι πολύ πιο αποδοτικός από τους μη βέλτιστους σε θέματα κόστους, αλλά οι μη βέλτιστοι αλγόριθμοι δεν είναι στον ίδιο βαθμό αποδοτικοί σε θέματα χρόνου. Έτσι, συμπεραίνουμε ότι από τους τρεις παράλληλους αλγόριθμους προθεματικού αθροίσματος που μελετήσαμε, γενικά ο πιο αποδοτικός είναι ο βέλτιστος αλγόριθμος. Στην περίπτωση όμως που για κάποια εφαρμογή δεν μας ενδιαφέρει πολύ το κόστος ή μας ενδιαφέρει περισσότερο η αποδοτικότητα του χρόνου εκτέλεσης, η καλύτερη επιλογή είναι ο μη βέλτιστος αλγόριθμος στο μοντέλο EREW PRAM, ο οποίος είναι ο γρηγορότερος.

Κεφάλαιο 7

Αλγόριθμος Παράλληλης Αναζήτησης σε Ταξινομημένη Λίστα

7.1 Πρόβλημα

Τα δεδομένα στο πρόβλημα της αναζήτησης είναι μια ταξινομημένη λίστα η οποία περιέχει n αριθμούς. Επίσης, ως δεδομένο δίνεται και ένας αριθμός y ο οποίος αποτελεί το στοιχείο αναζήτησης. Δηλαδή, το y είναι η τιμή για την οποία πρέπει να ψάξουμε αν υπάρχει ίδια τιμή στη λίστα. Στην περίπτωση που η λίστα περιλαμβάνει μια τιμή ίση με το y, τότε ως αποτέλεσμα δίνεται η θέση στην οποία βρέθηκε η τιμή αυτή. Σε αντίθετη περίπτωση, επιστρέφεται η θέση της λίστας που περιέχει την αμέσως μικρότερη τιμή από την τιμή του y.

7.2 Σειριακός Αλγόριθμος

Ο σειριακός αλγόριθμος με τον οποίο επιλύουμε το πρόβλημα της αναζήτησης είναι ο γνωστός αλγόριθμος δυαδικής αναζήτησης.

Ο επεξεργαστής ο οποίος καλείται να προβεί σε υπολογισμούς, αρχικά βρίσκει τη μεσαία θέση της λίστας μ και έπειτα ελέγχει αν η τιμή στη θέση αυτή είναι ίση με την τιμή της μεταβλητής y. Στην περίπτωση που η τιμή είναι ίση με το y, τότε επιστρέφεται η θέση μ και ο αλγόριθμος τερματίζει την εκτέλεσή του. Σε αντίθετη περίπτωση,

συνεχίζει η εκτέλεση του προγράμματος με την πιο πάνω διαδικασία να επαναλαμβάνεται σε κάποιο μέρος της λίστας. Συγκεκριμένα, εάν η τιμή στη θέση μ είναι μικρότερη από την τιμή του y, τότε η διαδικασία επαναλαμβάνεται σε όλα τα στοιχεία που βρίσκονται μετά την θέση μ. Εάν η τιμή στη θέση μ είναι μεγαλύτερη από την τιμή του y, τότε η διαδικασία επαναλαμβάνεται σε όλα τα στοιχεία που βρίσκονται πριν την θέση μ. Η εκτέλεση του αλγορίθμου τερματίζεται όταν βρεθεί η τιμή y στη λίστα ή όταν προβούμε σε έλεγχο για όλες τις τιμές της λίστας και δεν βρεθεί η τιμή y. Σε αυτή την περίπτωση θα επιστρέψουμε την τελευταία θέση στην οποία κάναμε τον έλεγχο.

Ο σειριακή αλγόριθμος για την δυαδική αναζήτηση είναι ο ακόλουθος:

$while\ (y\ not\ found)$
$\mu = list\ size/2$
$if\ (x_\mu = y)$
$return\ \mu$
$else\ if\ (x_\mu < y)$
$Repeat\ for\ x_{\mu+1}, \dots, x_n$
$else\ if\ (x_\mu > y)$
$Repeat\ for\ x_1, \dots, x_{\mu-1}$
$end\ do$

Ο αλγόριθμος αυτός εκτελείται σε χρόνο $O(\log n)$. Η αλγόριθμος χρησιμοποιεί μια μέθοδο αναζήτησης κατά την οποία το μέγεθος του προβλήματος μειώνεται κάθε φορά στο μισό. Αφού το μέγεθος του προβλήματος μειώνεται στο μισό σε κάθε επανάληψη και το πλήθος όλων των δεδομένων εισόδου είναι n, ο συνολικός αριθμός των επαναλήψεων που εκτελούνται είναι $O(\log n)$ στην χειρότερη περίπτωση. Στη χειρότερη περίπτωση η τιμή της y δεν βρίσκεται στη λίστα και έτσι η επανάληψη εκτελείται μέχρι το τέλος. Σε αντίθετη περίπτωση ο αριθμός των επαναλήψεων είναι μικρότερος. Στην περίπτωση που η τιμή αναζήτησης βρίσκεται στη μεσαία θέση της λίστας αναζήτησης, τότε ο αλγόριθμος εκτελείται σε χρόνο $\Theta(1)$. Οι πράξεις στο εσωτερικό της επανάληψης μπορούν να εκτελεστούν σε σταθερό χρόνο, $\Theta(1)$. Άρα, ο συνολικός χρόνος εκτέλεσης του σειριακού αλγόριθμου αναζήτησης είναι $O(\log n)$.

7.3 Πρώτος Παράλληλος Αλγόριθμος

Ο πρώτος παράλληλος αλγόριθμος αναζήτησης που θα περιγράψουμε είναι η παραλληλοποίηση του αλγορίθμου δυαδικής αναζήτησης. Τα n στοιχεία της ταξινομημένης λίστας είναι αποθηκευμένα στη κοινόχρηστη μνήμη, το ίδιο και η μεταβλητή y. Για την υλοποίηση του αλγορίθμου αυτού η λίστα χωρίζεται σε P υπολίστες, όπου P είναι ο αριθμός των επεξεργαστών που χρησιμοποιούμε για την επίλυση του προβλήματος. Ο αριθμός των επεξεργαστών που χρησιμοποιούνται είναι πάντα μικρότερος από το πλήθος των δεδομένων. Η κάθε μια από τις P υπολίστες περιέχει n/P ταξινομημένα στοιχεία και ο κάθε επεξεργαστής αναλαμβάνει μια υπολίστα για να προβεί σε υπολογισμούς.

Ο αλγόριθμος αυτός λειτουργεί με τον εξής τρόπο: ο κάθε επεξεργαστής τρέχει τον σειριακό αλγόριθμο δυαδικής αναζήτησης στην υπολίστα που του αντιστοιχεί. Όλοι οι επεξεργαστές τρέχουν παράλληλα στην υπολίστα τους τον σειριακό αλγόριθμο. Θεωρούμε ότι στον κάθε επεξεργαστή αντιστοιχεί μια υπολίστα ανάλογα με το αναγνωριστικό του επεξεργαστή. Ο πρώτος επεξεργαστής αναλαμβάνει την πρώτη υπολίστα, ο δεύτερος την δεύτερη, ο $p_{οστός}$ την τελευταία υπολίστα.

Στην περίπτωση που η τιμή της μεταβλητής y δεν βρέθηκε σε κάποια υπολίστα μετά την εκτέλεση του πιο πάνω αλγόριθμου, εκτελείται ένα επιπλέον βήμα. Στο βήμα αυτό γίνεται έλεγχος από τους επεξεργαστές εάν η τιμή της μεταβλητής y είναι μεταξύ των ακραίων στοιχείων δύο συνεχόμενων υπολιστών. Για παράδειγμα, για υπολίστες μεγέθους n/P θα γίνει έλεγχος αν το y είναι μεταξύ του στοιχείου στη θέση $i * (n/P)$ και $i * (n/P) + 1$, όπου i είναι αριθμός ίσος με το αναγνωριστικό του κάθε επεξεργαστή που εκτελεί την πράξη. Στη περίπτωση που κάτι τέτοιο ισχύει η θέση της τιμής y ισούται με $i * (n/P)$. Για την εκτέλεση του βήματος αυτού χρησιμοποιούνται $P - 1$ επεξεργαστές, αφού ο επεξεργαστής που εκτελεί υπολογισμούς για την τελευταία υπολίστα δεν χρειάζεται να προβεί στον έλεγχο που αναφέρθηκε.

Ο αλγόριθμος αυτός είναι υλοποιημένος στο μοντέλο CREW PRAM, δηλαδή υπάρχει ταυτόχρονη ανάγνωση της ίδιας κυψελίδας μνήμης. Σε κάθε παράλληλη εκτέλεση του

σειριακού αλγορίθμου δυαδικής αναζήτησης οι επεξεργαστές διαβάζουν όλοι ταυτόχρονα την τιμή y από την κοινόχρηστη μνήμη ώστε να προβούν στις απαραίτητες συγκρίσεις της αναζήτηση.

Ο πρώτος παράλληλος αλγόριθμος για την αναζήτηση ενός στοιχείου είναι ο ακόλουθος:

Βήμα 1:

$Processors\ i = 1\ to\ P\ Run\ Binary\ Search\ Algorithm$

$On(\ x_{(i-1)(n/p)+1}, .., x_{i(n/p)}\ )$

Βήμα 2:

$Processors\ i = 1\ to\ P - 1$

$if\ (x_{i*(n/P)} < y < x_{i*(n/P)+1})$

$return\ i * (n/P)$

$end\ do$

Μέρος του αλγόριθμου αυτού είναι όπως αναφέραμε η παραλληλοποίηση του σειριακού αλγόριθμου αναζήτησης ο οποίος εκτελείται σε χρόνο $\Theta(\log n)$ για n δεδομένα εισόδου. Αφού στην περίπτωσή μας χρησιμοποιούμε τον αλγόριθμο για n/P δεδομένα εισόδου τότε είναι λογικό ότι ο χρόνος εκτέλεσης του πρώτου βήματος του παράλληλου αλγορίθμου αναζήτησης είναι $O(\log n/P)$. Το δεύτερο βήμα εκτελείται σε σταθερό χρόνο, οπόταν δεν επηρεάζει τον ασυμπτωτικό χρόνο εκτέλεσης και έτσι συνολικά ο αλγόριθμος που περιγράφτηκε εκτελείται σε χρόνο $O(\log n/P)$.

Ο παράλληλος αλγόριθμος αυτός χρησιμοποιεί για την εκτέλεση του P επεξεργαστές, όπου $P \leq n$. Άρα, αφού ο θεωρητικός χρόνος εκτέλεσης του είναι $O(\log n/P)$ το κόστος εκτέλεσης είναι $P * \Theta(\log n/P)$. Επομένως, η πολυπλοκότητα κόστους του πρώτου αλγορίθμου παράλληλης αναζήτησης είναι $O(P \log n/P)$.

Συνοψίζοντας, η πολυπλοκότητα χρόνου του παράλληλου αλγόριθμου δυαδικής αναζήτησης είναι $O(\log n/P)$ και η πολυπλοκότητα κόστους είναι $O(P \log n/P)$. Όπως φαίνεται τόσο ο χρόνος όσο και το κόστος εξαρτάτε εκτός από το πλήθος των

δεδομένων εισόδου και από το πλήθος των επεξεργαστών που έχουμε στη διάθεση μας για την επίλυση του προβλήματος.

7.4 Δεύτερος Παράλληλος Αλγόριθμος

Τα δεδομένα του προβλήματος για τον αλγόριθμο αυτό, δηλαδή τα n στοιχεία της ταξινομημένης λίστας και η μεταβλητή y που είναι η τιμή αναζήτησης, είναι αποθηκευμένα στη κοινόχρηστη μνήμη.

Ο αλγόριθμος αυτός αποτελεί μια γενικευμένη παράλληλη εκδοχή της δυαδικής αναζήτησης, σε αντίθεση με τον προηγούμενο αλγόριθμο που απλά εκτελούσε τη σειριακή αναζήτηση παράλληλα για διαφορετικά δεδομένα. Συγκεκριμένα, αντί δυαδική αναζήτηση εκτελούμε $(P + 1)$ αναζήτηση με τη χρήση P επεξεργαστών.

Αρχικά, όπως και σε κάθε επόμενο βήμα, η λίστα με τα δεδομένα χωρίζεται σε $P + 1$ συνεχόμενες και ταξινομημένες υπολίστες. Η κάθε υπολίστα περιέχει $n/(P + 1)$ στοιχεία με εξαίρεση σε ορισμένες περιπτώσεις την τελευταία υπολιστα. Ο κάθε επεξεργαστής αναλαμβάνει μια από τις P πρώτες υπολίστες και ελέγχει το τελευταίο της στοιχείο, έστω s_j. Στην περίπτωση που το τελευταίο στοιχείο είναι ίσο με την μεταβλητή y τότε έχει βρεθεί η λύση του προβλήματος και ο αλγόριθμος τερματίζει. Σε αντίθετη περίπτωση, ελέγχουμε αν το τελευταίο στοιχείο της υπολίστας είναι μεγαλύτερο ή μικρότερο από την τιμή του y. Εάν το τελευταίο στοιχείο s_j είναι μεγαλύτερο από το y, τότε αγνοούμε όλα τα στοιχεία τα οποία βρίσκονται μετά από αυτό για την συνέχεια του αλγορίθμου. Εάν το s_j είναι μεγαλύτερο από το y, τότε αγνοούμε όλα τα στοιχεία της λίστας που βρίσκονται πριν από το s_j συμπεριλαμβανομένου. Η διαδικασία αυτή επαναλαμβάνεται είτε μέχρι να βρεθεί το y στη λίστα, είτε μέχρι το μέγεθος της λίστας να είναι μικρότερο από τον αριθμό των επεξεργαστών. Όταν συμβαίνει η δεύτερη περίπτωση, εκτελείται παράλληλη σύγκριση των στοιχείων που απέμειναν, έτσι ώστε να επιστραφεί η σωστή θέση για το y.

Ουσιαστικά, σε αυτό τον αλγόριθμο όταν δεν γίνεται εύρεση του y από τον έλεγχο στο τελευταίο στοιχείο μιας υπολίστας, η πιο πάνω διαδικασία αναζήτησης μεταφέρεται σε ένα συγκεκριμένο πεδίο στοιχείων της λίστας.

Ο αλγόριθμος αυτός, όπως και ο προηγούμενος, υλοποιείται στο μοντέλο CREW PRAM. Όταν οι επεξεργαστές εκτελούν παράλληλα την αναζήτηση, διαβάζουν ταυτόχρονα την κοινόχρηστη μεταβλητή y για να προβούν στους ελέγχους που περιγράφηκαν προηγουμένως. Άρα, υπάρχει ταυτόχρονη πρόσβαση στην ίδια κυψελίδα της κοινόχρηστης μνήμης από περισσότερους από ένα επεξεργαστές.

Ο παράλληλος αλγόριθμος αναζήτησης εκτελείται σε χρόνο $O(\log n / \log P)$. Σε κάθε βήμα του αλγορίθμου ο διαχωρισμός των ταξινομημένων υπολιστών και η ανάθεση τους σε επεξεργαστές γίνεται σε σταθερό χρόνο $\Theta(1)$. Επιπλέον, οι έλεγχοι που απαιτούνται για την σύγκριση της τιμής y με το τελευταίο στοιχείο s_j της κάθε υπολίστας χρειάζονται και αυτοί σταθερό χρόνο. Οι πιο πάνω πράξεις που αναφέρθηκαν βρίσκονται στο εσωτερικό μιας επανάληψης και συνολικά απαιτούν σταθερό χρόνο. Σταθερός χρόνος απαιτείται και για την εύρεση της θέσης του y στη λίστα, στην περίπτωση που το y δεν υπάρχει στη λίστα.

Η επανάληψη του αλγορίθμου, στην οποία γίνονται οι πιο κύριες πράξεις του αλγορίθμου, μπορεί στην καλύτερη περίπτωση να εκτελεστεί σε σταθερό χρόνο, στην περίπτωση που βρεθεί το y από την αρχή. Στη περίπτωση που το y βρεθεί σε κάποια υπολίστα τότε η επανάληψη δεν εκτελείται μέχρι το τέλος, αλλά διακόπτεται μόλις βρεθεί τιμή ίση με το y. Στη χειρότερη περίπτωση, η επανάληψη εκτελείται μέχρι το μέγεθος της παρούσας υπολίστας να μην ξεπερνά τον αριθμό των επεξεργαστών. Έτσι, πρέπει να βρούμε τον αριθμό των επαναλήψεων που θα εκτελεστούν μέχρι το μέγεθος της λίστας να γίνει ίσο με τον αριθμό των επεξεργαστών. Μπορούμε να παρατηρήσουμε ότι μετά από την εκτέλεση μιας επανάληψης το μέγεθος ολόκληρης της παρούσας λίστας μειώνεται κατά $1/(P+1)$. Αυτό συμβαίνει αφού απομένει μόνο μια υπολίστα από τις $P+1$ υπολίστες της προηγούμενης επανάληψης, στην οποία πρέπει να συνεχιστεί η εκτέλεση του αλγορίθμου. Άρα, αφού η εκτέλεση του αλγορίθμου ξεκινά με μια λίστα από n στοιχεία και μετά από κάθε επανάληψη το μέγεθός της μειώνεται κατά $1/(P+1)$, στην τελευταία επανάληψη, έστω λ, το μέγεθος της θα είναι $n/(P+1)^{\lambda-1}$. Στον Πίνακα 7.1 μπορούμε να δούμε αναλυτικά πως καταλήγουμε σε αυτό το συμπέρασμα. Αφού, στη τελευταία επανάληψη το μέγεθος θα είναι $n/(P+1)^{\lambda-1}$ και γνωρίζουμε ότι η εκτέλεση της επανάληψης σταματά όταν το

μέγεθος της λίστας γίνει ίσο με το πλήθος των επεξεργαστών, τότε ισχύει ότι $n/(P+1)^{\lambda-1} = P+1$. Από αυτό καταλήγουμε στο $n = (P+1)\lambda$. Άρα, θα εκτελεστεί η τελευταία επανάληψη μετά από $O(\log n / \log P)$ επαναλήψεις [4, 5].

Επανάληψη	Μέγεθος Λίστας
1	n
2	$n/(p+1)$
3	$n/(p+1)^2$
…	…
i	$n/(p+1)^{i-1}$

Πίνακας 7.1 – Μεγέθη λίστας σε κάθε επανάληψη

Αφού, συμπεράναμε ότι ο χρόνος εκτέλεσης του αλγορίθμου είναι $O(\log n / \log P)$ και γνωρίζουμε ότι ο αριθμός των επεξεργαστών που χρησιμοποιούνται είναι ίσος με P, τότε το κόστος είναι $P * O(\log n / \log P)$. Επομένως, η πολυπλοκότητα κόστους του δεύτερου παράλληλου αλγορίθμου αναζήτησης είναι $O(P \log n / \log P)$.

Με βάση την πολυπλοκότητα του αλγορίθμου αυτού συμπεραίνουμε ότι τόσο ο χρόνος όσο και το κόστος εξαρτάται εκτός από το πλήθος των δεδομένων εισόδου και από το πλήθος των επεξεργαστών που έχουμε στη διάθεση μας για την επίλυση του προβλήματος.

Από την πολυπλοκότητα του κόστους μπορούμε να παρατηρήσουμε ότι ο αλγόριθμος αυτός δεν είναι βέλτιστος. Το κόστος του είναι μεγαλύτερο από την πολυπλοκότητα του καλύτερου σειριακού αλγορίθμου που επιλύει το πρόβλημα της αναζήτησης, που περιγράφηκε στην αρχή του κεφαλαίου. Σε αυτό το σημείο υπενθυμίζουμε ότι το κόστος του καλύτερου σειριακού αλγορίθμου είναι $\Theta(\log n)$, ενώ το κόστος εκτέλεσης του αλγορίθμου αυτού είναι $O(P \log n / \log P)$. Στην περίπτωση όμως που ο αριθμός των επεξεργαστών είναι σταθερός αριθμός σε σχέση με το πλήθος των δεδομένων εισόδου, ο αλγόριθμος γίνεται ασυμπτωτικά βέλτιστος. Αφού το P είναι σταθερό, το ίδιο ισχύει και για το $\log P$, άρα το κόστος του αλγόριθμου γίνεται $O(\log n)$.

Αν και ο αλγόριθμος δεν είναι πάντα βέλτιστος, ο χρόνος στον οποίο εκτελείται αποτελεί το κάτω φράγμα στην αναζήτηση. Δηλαδή, είναι αδύνατο να υλοποιηθεί κάποιος παράλληλος αλγόριθμος που να επιλύει το πρόβλημα της αναζήτησης σε χρόνο μικρότερο από O($\log n / \log P$).

7.5 Πειραματική Αξιολόγηση Αλγορίθμων

Οι δύο αλγόριθμοι παράλληλης αναζήτησης που μελετήσαμε υλοποιήθηκαν στο μοντέλο CREW PRAM με την χρήση της γλώσσας XMTC και προσομοιώθηκαν με την χρήση του XMT για να πάρουμε τα αποτελέσματα του παράλληλου υπολογισμού. Ακολουθούν οι μετρήσεις που πάρθηκαν από την προσομοίωση των δύο αλγορίθμων και τα συμπεράσματα στα οποία καταλήγουμε βάσει των μετρήσεων.

Οι μετρήσεις πάρθηκαν από διάφορες εκτελέσεις του κάθε αλγορίθμου με διαφορετικό αριθμό δεδομένων εισόδου n και επεξεργαστών P. Για το ίδιο πλήθος δεδομένων πάρθηκαν μετρήσεις της εκτέλεσης του προγράμματος με διαφορετικό αριθμό επεξεργαστών. Επιπλέον, οι ίδιες μετρήσεις έγιναν σε δύο φάσεις, στις οποίες κρατούμε σταθερό το πλήθος των δεδομένων και των επεξεργαστών, αλλά μεταβάλλεται η τιμή του αριθμού που αναζητούμε, δηλαδή του y. Στην μια φάση η τιμή του y υπάρχει στην λίστα με τα δεδομένα εισόδου, ενώ στην άλλη δεν υπάρχει. Για την κάθε μια από τις ξεχωριστές περιπτώσεις που υπάρχουν λήφθηκαν μετρήσεις για διαφορετικά δεδομένα από τα οποία υπολογίστηκε μια μέση τιμή χρόνου, η οποία παρουσιάζεται πιο κάτω.

Τα δεδομένα του προβλήματος είναι ταξινομημένοι ακέραιοι αριθμοί σε αύξουσα σειρά και βρίσκονται σε ένα αρχείο που δημιουργημένο από τον προγραμματιστή. Το αρχείο αυτό διαβάζεται με τη βοήθεια του Εργαλείου Μνήμης που παρέχει η XMTC. Οι τιμές ενσωματώνονται στο πρόγραμμα με την εισαγωγή του header αρχείου, που δημιουργήθηκε από το Εργαλείο Μνήμης, στον κώδικα του προγράμματος.

Πρώτος Αλγόριθμος Παράλληλης Αναζήτησης

Στους ακόλουθους πίνακες μπορούμε να δούμε τον χρόνο (msec) και το κόστος εκτέλεσης του πρώτου αλγορίθμου παράλληλης αναζήτησης με n δεδομένα εισόδου και

με τη χρήση P επεξεργαστών. Επίσης, στον πίνακες φαίνεται ο χρόνος ($\log n/P$) που θεωρητικά πρέπει να έχει ο αλγόριθμος με βάση την ασυμπτωτικής ανάλυσης.

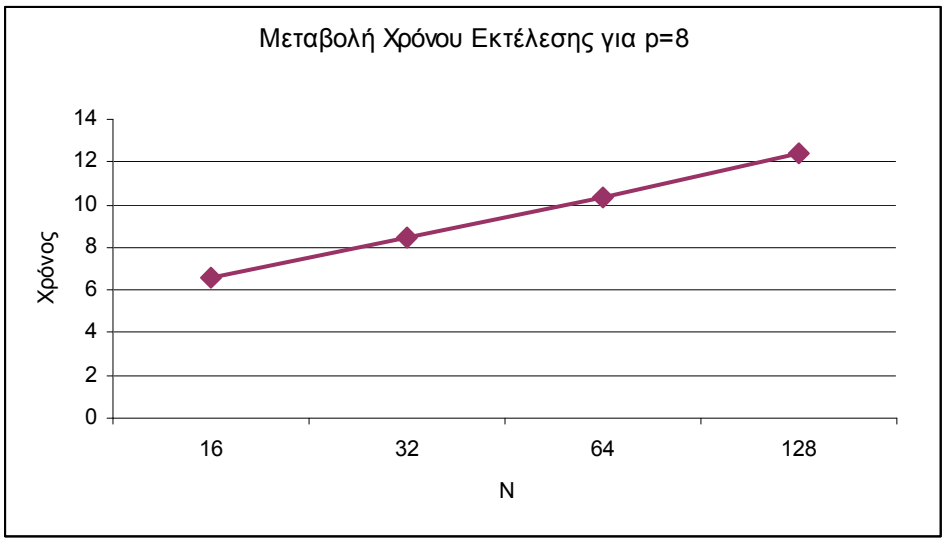
Ο Πίνακας 7.2 περιέχει τα αποτελέσματα των μετρήσεων που έγιναν στις περιπτώσεις που η τιμή αναζήτησης είναι ανάμεσα στα δεδομένα εισόδου, ενώ ο Πίνακας 7.3 περιέχει τα αποτελέσματα για τις περιπτώσεις που η τιμή αναζήτησης δεν είναι ίση με καμία τιμή από τα δεδομένα.

n	**P**	**T = log(n/P)**	**Χρόνος**	**Κόστος**
16	4	2	8.51	34.04
	8	1	6.59	52.72
32	4	3	10.12	40.48
	8	2	8.41	67.28
	16	1	6.67	106.72
64	8	3	10.38	83.04
	16	2	8.87	141.92
	32	1	6.83	218.56
128	8	4	12.37	98.96
	16	3	9.11	145.76
	32	2	7.22	231.04

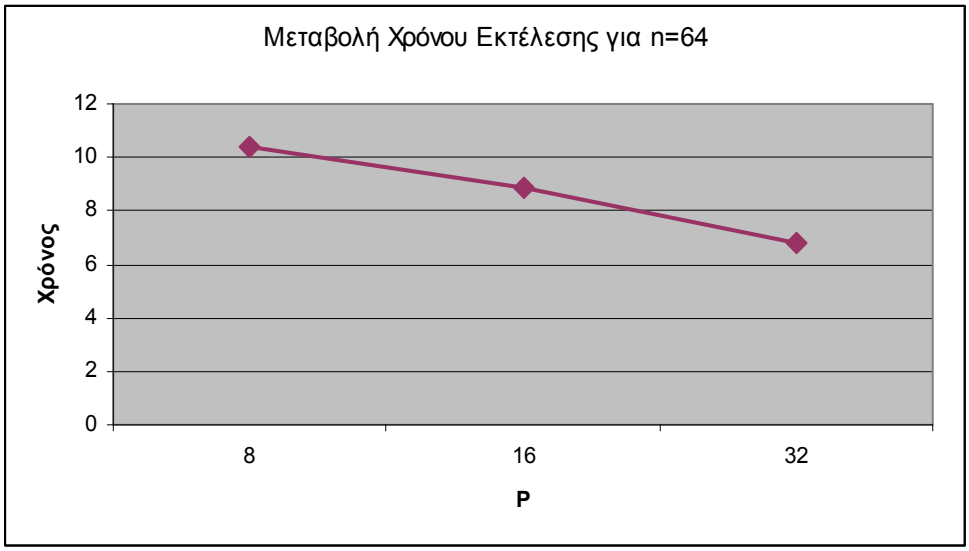
Πίνακας 7.2 – Μετρήσεις Προσομοίωσης για την περίπτωση που η τιμή αναζήτησης είναι ανάμεσα στα δεδομένα εισόδου

Στον πιο πάνω πίνακα μπορούμε να δούμε την μεταβολή του χρόνου και του κόστους για κάθε συνδυασμό πλήθους δεδομένων εισόδου και πλήθους επεξεργαστών. Αρχικά, παρατηρούμε ότι διατηρώντας σταθερό τον αριθμό των επεξεργαστών και αυξάνοντας τον αριθμό των δεδομένων εισόδου, ο χρόνος εκτέλεσης του αλγορίθμου αυξάνεται. Το γεγονός αυτό είναι λογικό αφού ο ασυμπτωτικός χρόνος εκτέλεσης του αλγορίθμου αναζήτησης εξαρτάται από το πλήθος των δεδομένων εισόδου. Επιπλέον, όπως συμπεράνουμε και από την Γραφική 7.1, η αύξηση που παρατηρείται είναι λογαριθμική. Το ίδιο συμβαίνει και για το κόστος εκτέλεσης του αλγορίθμου, το οποίο

όπως είναι φυσικό αυξάνεται με την αύξηση του αριθμού δεδομένων εισόδου και του χρόνου εκτέλεσης.



Γραφική 7.1 – Μεταβολή χρόνου εκτέλεσης για p=8 όταν η τιμή αναζήτησης είναι ανάμεσα στα δεδομένα εισόδου



Γραφική 7.2 – Μεταβολή χρόνου εκτέλεσης για n=64 όταν η τιμή αναζήτησης είναι ανάμεσα στα δεδομένα εισόδου

Στην Γραφική 7.2 φαίνεται η μεταβολή του χρόνου εκτέλεσης όταν διατηρούμε σταθερό το πλήθος των δεδομένων εισόδου (n=64) και μεταβάλλουμε τον αριθμό των επεξεργαστών. Παρατηρούμε ότι ο χρόνος εκτέλεσης του αλγορίθμου μειώνεται λογαριθμικά καθώς αυξάνεται ο αριθμός των επεξεργαστών. Το φαινόμενο αυτό συμβαίνει γιατί ο ασυμπτωτικός χρόνος εκτέλεσης του αλγορίθμου αυτού εκτός από τον αριθμό των δεδομένων εισόδου, εξαρτάται από τον αριθμό των επεξεργαστών.

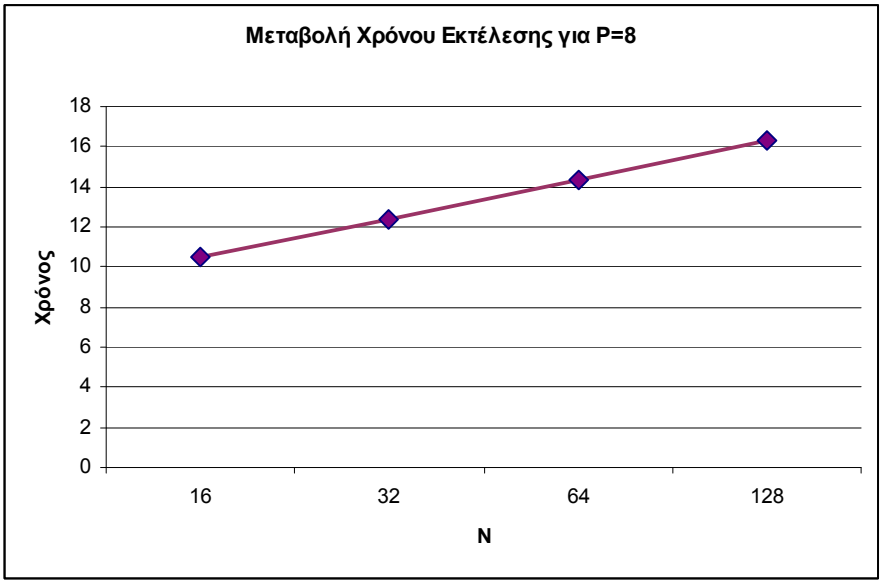
Επαναλαμβάνοντας, το ίδιο για το κόστος παρατηρούμε ότι το κόστος αυξάνεται καθώς αυξάνεται ο αριθμός των επεξεργαστών. Το γεγονός αυτό είναι λογικό αφού το κόστος είναι ανάλογο του αριθμού των επεξεργαστών και έτσι, προκαλείται αύξηση του κόστους όταν οι επεξεργαστές αυξάνονται.

Στη συνέχεια παρατηρούμε τη σχέση μεταξύ του θεωρητικού χρόνου και του πραγματικού χρόνου εκτέλεσης του αλγορίθμου. Όπως βλέπουμε, ο πραγματικός χρόνος που χρειάζεται για να ολοκληρωθεί η εκτέλεση του προγράμματος είναι μεγαλύτερος από τον θεωρητικό χρόνο. Παρόλα αυτά, ο πραγματικός χρόνος αυξάνεται με σταθερό ρυθμό, περίπου κατά 2ms σε κάθε αύξηση του αριθμού των δεδομένων εισόδου. Η διαφορά που παρατηρείται μεταξύ του θεωρητικού και πραγματικού χρόνου είναι δικαιολογημένη αφού ο θεωρητικός χρόνος είναι ασυμπτωτικός, δηλαδή δεν λαμβάνει υπόψη του ένα σύνολο από σταθερές. Αν και αυτές οι σταθερές δεν επηρεάζουν τον ασυμπτωτικό χρόνο εκτέλεσης, παίζουν ρόλο στον καθορισμό του πραγματικού χρόνου εκτέλεσης. Επιπρόσθετα, λόγω του ότι ο αλγόριθμος είναι υλοποιημένος στο μοντέλο CREW PRAM, υπάρχει μια επιπλέον καθυστέρηση στο χρόνο εκτέλεσης του προγράμματος αφού παρατηρείται ταυτόχρονη ανάγνωση της ίδιας κοινόχρηστης κυψελίδας που απαιτεί κάποιο επιπλέον χρόνο.

n	**P**	**T = $\log(n/P)$**	**Χρόνος**	**Κόστος**
16	4	2	12.35	49.4
	8	1	10.52	84.16
32	4	3	13.22	52.88
	8	2	12.36	98.88
	16	1	10.63	170.08
64	8	3	14.34	114.72
	16	2	12.44	199.04
	32	1	10.67	341.44
128	8	4	16.35	130.8
	16	3	14.43	230.88
	32	2	12.61	403.52

Πίνακας 7.3 – Μετρήσεις Προσομοίωσης για την περίπτωση που η τιμή αναζήτησης δεν είναι ανάμεσα στα δεδομένα εισόδου

Στον πιο πάνω πίνακα, περιέχονται οι μετρήσεις που λήφθηκαν στην περίπτωση που η τιμή αναζήτησης για την οποία πρέπει να επιστραφεί η θέση της στη λίστα εισόδου, δεν υπάρχει στην λίστα. Παρατηρώντας τις μετρήσεις που πάρθηκαν σε αυτή την περίπτωση, καταλήγουμε στα ίδια συμπεράσματα που αναφέρθηκαν προηγουμένως.
Στην πιο κάτω γραφική παράσταση μπορούμε να δούμε την μεταβολή του χρόνου όταν αυξάνεται ο αριθμός των δεδομένων εισόδου στην περίπτωση που χρησιμοποιούνται οχτώ επεξεργαστές για την επίλυση του προβλήματος.



Γραφική 7.3 – Μεταβολή χρόνου εκτέλεσης όταν η τιμή αναζήτησης δεν είναι ανάμεσα στα δεδομένα εισόδου

Από την πιο πάνω γραφική παράσταση, επιβεβαιώνεται η παρατήρηση που έγινε ότι ο χρόνος εκτέλεσης του αλγορίθμου αυξάνεται με την αύξηση του αριθμού των δεδομένων εισόδου. Παρατηρούμε, ότι κάθε φορά που αυξάνεται ο αριθμός των δεδομένων, ο χρόνος αυξάνεται σταθερά περίπου κατά 2ms. Άρα, η αύξηση του χρόνου σε σχέση με το πλήθος των δεδομένων είναι λογαριθμική.

Συγκρίνοντας, τις μετρήσεις που λήφθηκαν στις δύο περιπτώσεις παρατηρούμε ότι στη δεύτερη περίπτωση, τόσο ο χρόνος όσο και το κόστος έχουν μεγαλύτερες τιμές. Δηλαδή, ο πρώτος αλγόριθμος αναζήτησης καθυστερεί περισσότερο να βρει λύση στην περίπτωση που η τιμή που αναζητείται δεν υπάρχει ανάμεσα στα δεδομένα εισόδου. Αυτό δικαιολογείται αφού σε αυτό τον αλγόριθμο ο κάθε επεξεργαστής εκτελεί

παράλληλα τον σειριακό αλγόριθμο δυαδικής αναζήτησης. Σε αυτό τον αλγόριθμο, πρέπει να γίνουν όλοι οι πιθανοί έλεγχοι για την επιβεβαίωση ότι η τιμή δεν περιλαμβάνεται στη λίστα και στο τέλος να βρεθεί η θέση της τιμής αναζήτησης στη λίστα. Ως αποτέλεσμα, στην περίπτωση αυτή ο αλγόριθμος φτάνει στο χείριστο αριθμό επαναλήψεων, σε αντίθεση με την πρώτη περίπτωση όπου ο αλγόριθμος σταματά την εκτέλεση των επαναλήψεων μόλις βρεθεί λύση. Για αυτό το λόγο, ο χρόνος εκτέλεσης του αλγορίθμου στη δεύτερη περίπτωση είναι μεγαλύτερος από τον χρόνο εκτέλεσης στην περίπτωση που η τιμή αναζήτησης είναι ανάμεσα στα στοιχεία της λίστας.

Δεύτερος Αλγόριθμος Παράλληλης Αναζήτησης

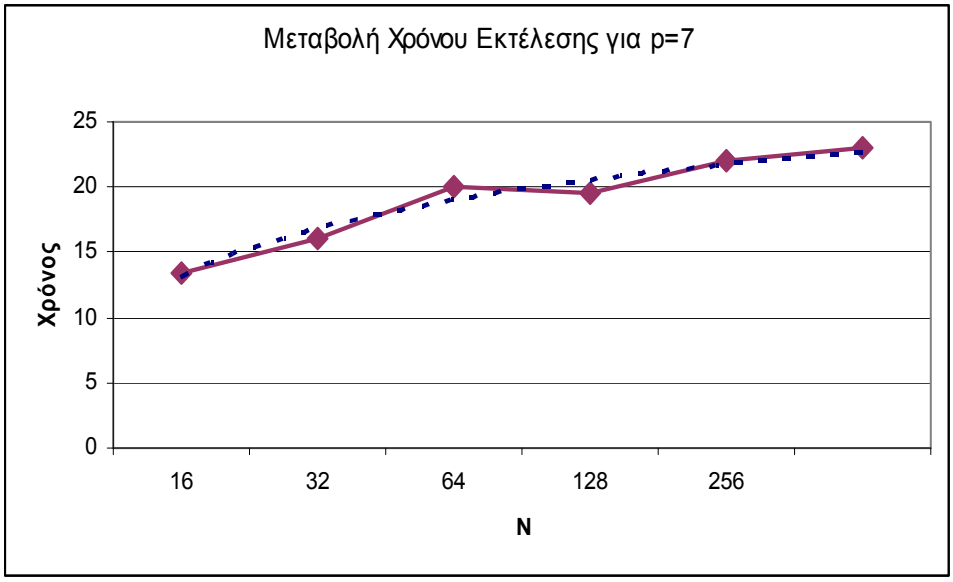
Στους ακόλουθους πίνακες μπορούμε να δούμε τον χρόνο και το κόστος εκτέλεσης του δεύτερου αλγορίθμου παράλληλης αναζήτησης με n δεδομένα εισόδου και με τη χρήση P επεξεργαστών. Επίσης, στον πίνακες φαίνεται ο χρόνος $(\log n / \log P)$ που θεωρητικά πρέπει να έχει ο αλγόριθμος με βάση την ασυμπτωτικής ανάλυσης.

Ο Πίνακας 7.4 περιέχει τα αποτελέσματα των μετρήσεων που έγιναν στις περιπτώσεις που η τιμή αναζήτησης είναι ανάμεσα στα δεδομένα εισόδου, ενώ ο Πίνακας 7.5 περιέχει τα αποτελέσματα για τις περιπτώσεις που η τιμή αναζήτησης δεν είναι ίση με καμία τιμή από τα δεδομένα.

Για κάθε περίπτωση, παρουσιάζεται μια γραφική παράσταση που αναπαριστά τη μεταβολή του χρόνου εκτέλεσης στη περίπτωση που ο αριθμός των επεξεργαστών παραμένει σταθερός.

n	P	$T = \log n / \log P$	Χρόνος	Κόστος
8	3	1.90	15.5	46.5
	5	1.29	14.45	72.25
	7	1.07	13.43	94.01
16	5	1.72	18.49	92.45
	7	1.42	16.06	112.42
	9	1.26	14.52	130.68
	13	1.08	13.5	175.5
32	5	2.15	21.42	107.1
	7	1.78	20.57	143.99
	11	1.45	18.01	198.11
	13	1.35	18.16	236.08
64	5	2.58	20.95	104.75
	7	2.14	19.43	136.01
	11	1.73	20.28	223.08
	13	1.62	20.28	263.64
128	7	2.49	21.81	152.67
	9	2.21	21.17	190.53
	11	2.02	18.49	203.39
256	7	2.85	22.3	156.10
	9	2.52	19.47	175.23
	11	2.31	19.44	253.72

Πίνακας 7.4 – Μετρήσεις Προσομοίωσης για την περίπτωση που η τιμή αναζήτησης είναι ανάμεσα στα δεδομένα εισόδου



Γραφική 7.4 – Μεταβολή χρόνου εκτέλεσης όταν η τιμή αναζήτησης είναι ανάμεσα στα δεδομένα εισόδου

Στον Πίνακα 7.4 βλέπουμε για κάθε πλήθος δεδομένων πως μεταβάλλεται η τιμή του χρόνου και του κόστους για διαφορετικό αριθμό επεξεργαστών. Στη πιο πάνω γραφική παράσταση αναπαρίσταται η μεταβολή του χρόνου εκτέλεσης σε σχέση με τα δεδομένα εισόδου στην περίπτωση που χρησιμοποιούνται εφτά επεξεργαστές. Αρχικά, παρατηρούμε ότι για ίδιο αριθμό επεξεργαστών ο χρόνος εκτέλεσης του αλγορίθμου αυξάνεται λογαριθμικά, καθώς αυξάνεται το πλήθος των δεδομένων. Η λογαριθμική αύξηση που παρατηρείται οφείλεται στο ότι ο ασυμπτωτικός χρόνος εκτέλεσης του αλγορίθμου αναζήτησης εξαρτάται από το πλήθος των δεδομένων εισόδου και είναι της τάξης $\Theta(\log n)$.

Επιπλέον, για σταθερό πλήθος δεδομένων παρατηρούμε ότι τις περισσότερες φορές καθώς αυξάνεται το πλήθος των επεξεργαστών, ο χρόνος εκτέλεσης του αλγορίθμου μειώνεται. Το γεγονός αυτό οφείλεται στο ότι ο ασυμπτωτικός χρόνος εκτέλεσης του αλγορίθμου είναι λογαριθμικά αντιστρόφως ανάλογος του αριθμού των επεξεργαστών. Έτσι, είναι αναμενόμενο για μεγαλύτερο αριθμό επεξεργαστών να χρειάζεται λιγότερος χρόνος για την εκτέλεση του αλγορίθμου. Η ίδια κατάσταση όμως δεν παρουσιάζεται και στην περίπτωση του κόστους. Όταν διατηρούμε σταθερό το πλήθος δεδομένων αλλά αυξάνουμε τον αριθμό των επεξεργαστών, το κόστος εκτέλεσης του αλγορίθμου αυξάνεται. Το φαινόμενο αυτό μπορεί να φαίνεται παράξενο αφού το κόστος εξαρτάται

από τον χρόνο ο οποίος μειώνεται με την αύξηση των επεξεργαστών. Είναι όμως δικαιολογημένο, αφού το κόστος εξαρτάται και από τον αριθμό των επεξεργαστών. Εξάλλου, στις περισσότερες περιπτώσεις η διαφορά του χρόνου εκτέλεσης ανάμεσα σε μετρήσεις με διαφορετικό αριθμό επεξεργαστών είναι σχετικά μικρή.

Σε αυτό το σημείο ας παρατηρήσουμε τη σχέση μεταξύ του θεωρητικού χρόνου εκτέλεσης και του πραγματικού χρόνου εκτέλεσης του αλγορίθμου. Όπως βλέπουμε, ο πραγματικός χρόνος που χρειάζεται για να ολοκληρωθεί η εκτέλεση του προγράμματος είναι μεγαλύτερος από τον θεωρητικό χρόνο. Το φαινόμενο αυτό που παρατηρείται είναι δικαιολογημένο αφού ο θεωρητικός χρόνος είναι ασυμπτωτικός, δηλαδή δεν λαμβάνει υπόψη του ένα σύνολο από σταθερές. Αν και αυτές οι σταθερές δεν επηρεάζουν τον ασυμπτωτικό χρόνο εκτέλεσης, παίζουν ρόλο στον καθορισμό του πραγματικού χρόνου εκτέλεσης. Επιπλέον, όπως αναφέρθηκε στην περιγραφή του αλγορίθμου αυτού, η υλοποίηση του αλγορίθμου γίνεται στο μοντέλο CREW PRAM. Το γεγονός αυτό έχει ως αποτέλεσμα μια επιπλέον καθυστέρηση στην εκτέλεση του προγράμματος αφού υπάρχει ταυτόχρονη ανάγνωση της ίδιας κοινόχρηστης κυψελίδας που απαιτεί κάποιο επιπλέον χρόνο.

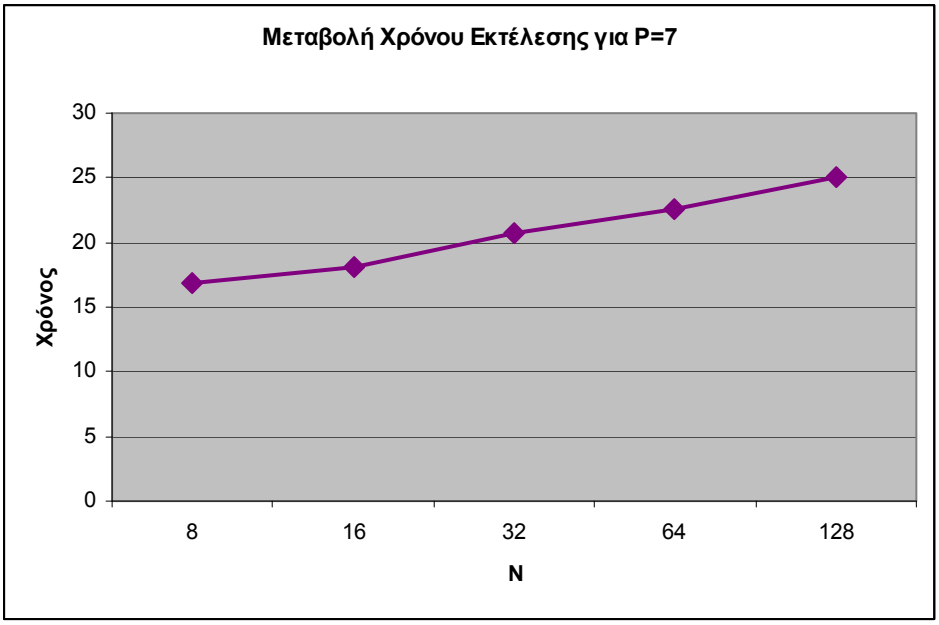
Η ίδια σχέση που παρουσιάζεται μεταξύ του θεωρητικού και του πραγματικού χρόνου εκτέλεσης, παρουσιάζεται και για το κόστος. Είναι όμως αναμενόμενο, αφού το θεωρητικό κόστος το οποίο δηλώνεται και αυτό ασυμπτωτικά, υπολογίζεται με βάση τον πραγματικό χρόνο εκτέλεσης.

n	P	$T = \log n / \log P$	Χρόνος	Κόστος
8	3	1.90	16.98	50.94
	5	1.29	16.34	81.7
	7	1.07	16.85	117.95
16	5	1.72	36.11	180.55
	7	1.42	18.08	126.56
	9	1.26	17.36	156.24
	13	1.08	16.93	220.09
32	5	2.15	35.87	179.35
	7	1.78	22.1	140.7
	11	1.45	20.33	223.63
	13	1.35	18.74	243.62
64	5	2.58	25.32	126.6
	7	2.14	22.51	157.57
	11	1.73	21.08	231.88
	13	1.62	18.91	245.83
128	7	2.49	25.09	175.63
	9	2.21	24.8	223.2
	11	2.02	24.67	271.37
256	7	2.85	23.05	161.35
	9	2.52	23.28	209.52
	11	2.31	22.55	293.15

Πίνακας 7.5 – Μετρήσεις Προσομοίωσης για την περίπτωση που η τιμή αναζήτησης δεν είναι ανάμεσα στα δεδομένα εισόδου

Ο πιο πάνω πίνακας περιέχει ακριβώς τις ίδιες περιπτώσεις μετρήσεων που λήφθηκαν και περιλαμβάνονται στον Πίνακα 7.4, αλλά η τιμή που αναζητείται είναι διαφορετική. Συγκεκριμένα, στον πίνακα αυτό περιλαμβάνονται μόνο περιπτώσεις στις οποίες στα δεδομένα εισόδου δεν βρίσκεται τιμή ίση με τη τιμή αναζήτησης. Παρατηρώντας, τις μετρήσεις που πάρθηκαν σε αυτή την περίπτωση, καταλήγουμε στα ίδια συμπεράσματα που αναφέρθηκαν προηγουμένως.

Στην πιο κάτω γραφική παράσταση μπορούμε να δούμε την μεταβολή του χρόνου όταν αυξάνεται ο αριθμός των δεδομένων εισόδου στην περίπτωση που χρησιμοποιούνται εφτά επεξεργαστές για την επίλυση του προβλήματος.



Γραφική 7.5 – Μεταβολή χρόνου εκτέλεσης όταν η τιμή αναζήτησης δεν είναι ανάμεσα στα δεδομένα εισόδου

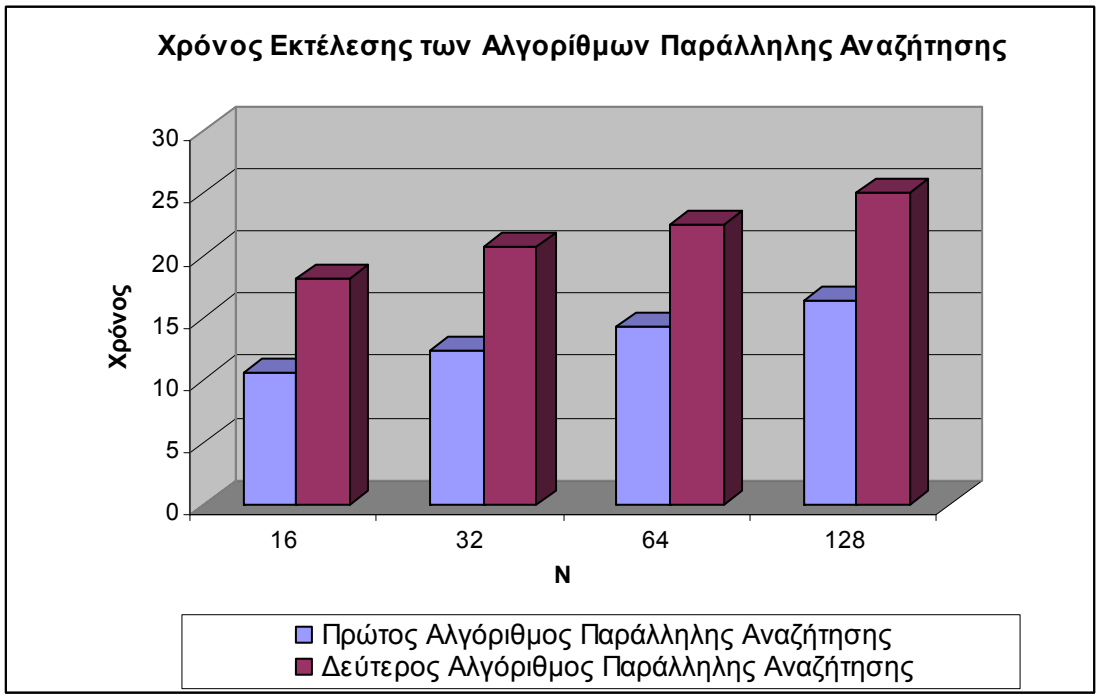
Από την πιο πάνω γραφική παράσταση, επιβεβαιώνεται ότι ο χρόνος εκτέλεσης του αλγορίθμου εξαρτάται από το πλήθος των δεδομένων και συγκεκριμένα αυξάνεται όταν αυξάνονται τα δεδομένα.

Συγκρίνοντας, το χρόνο και το κόστος εκτέλεσης που λήφθηκαν και στις δύο περιπτώσεις παρατηρούμε ότι στη δεύτερη περίπτωση, τόσο ο χρόνος όσο και το κόστος έχουν μεγαλύτερες τιμές. Δηλαδή, ο αλγόριθμος καθυστερεί περισσότερο να βρει λύση στο πρόβλημα αναζήτησης στην περίπτωση που η τιμή που αναζητείται δεν υπάρχει ανάμεσα στα δεδομένα εισόδου. Αυτό οφείλεται στο γεγονός ότι ο αλγόριθμος συνεχίζει να εκτελείται ψάχνοντας για την τιμή αναζήτησης μέχρι να φτάσει το τελικό σημείο όπου οι επεξεργαστές βρίσκουν τη θέση της στη λίστα με τα δεδομένα εισόδου. Άρα, στην περίπτωση που η τιμή αναζήτησης δεν βρίσκεται στη λίστα με τα δεδομένα εισόδου ο αλγόριθμος εκτελεί τον μεγαλύτερο αριθμό επαναλήψεων που θα μπορούσε. Για αυτό το λόγο, ο χρόνος εκτέλεσης του αλγορίθμου σε αυτές τις περιπτώσεις είναι

μεγαλύτερος από τον χρόνο εκτέλεσης στην περίπτωση που η τιμή αναζήτησης είναι ανάμεσα στα στοιχεία της λίστας.

7.6 Σύγκριση Αλγορίθμων

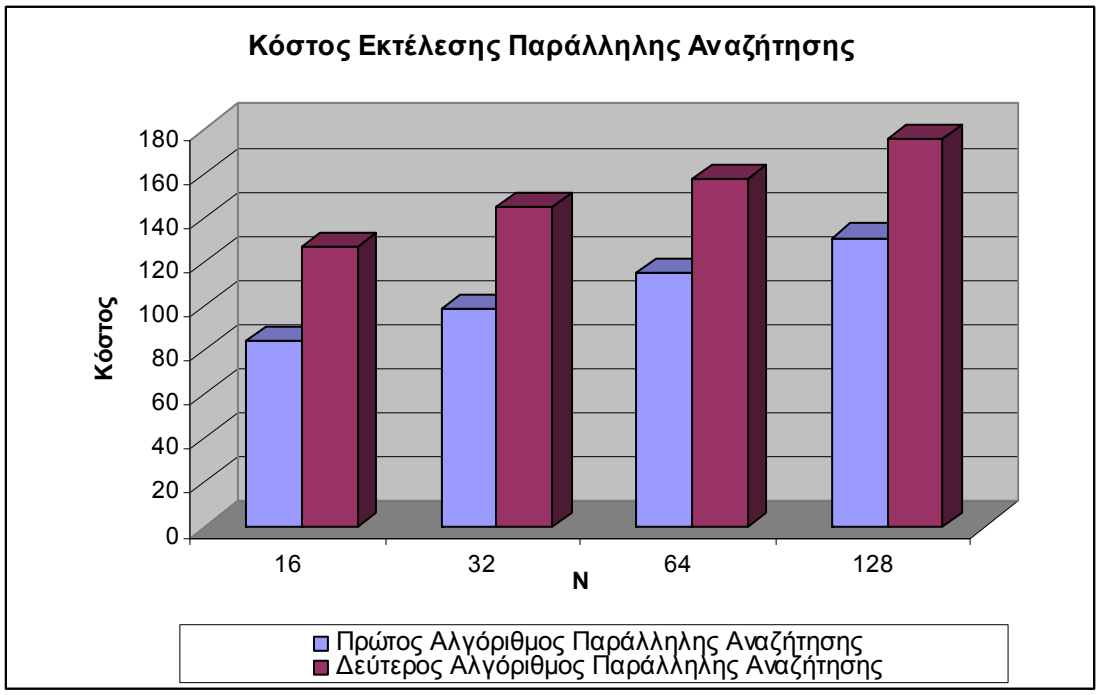
Στη συνέχεια ακολουθεί η σύγκριση των δύο αλγορίθμων παράλληλης αναζήτησης που περιγράφτηκαν και αξιολογήθηκαν ξεχωριστά προηγουμένως. Για καλύτερη σύγκριση των αλγορίθμων παρουσιάζονται πιο κάτω δύο γραφικές παραστάσεις. Τα αποτελέσματα που παρουσιάζονται για τον πρώτο αλγόριθμο αφορούν τις μετρήσεις για εκτελέσεις αλγορίθμων με την χρήση οχτώ επεξεργαστών, ενώ για τον δεύτερο εφτά επεξεργαστών.



Γραφική 7.6 – Σύγκριση Αλγορίθμων Παράλληλης Αναζήτησης

Από την πιο πάνω γραφική παράσταση παρατηρούμε την διαφορά στο χρόνο εκτέλεσης των δύο αλγορίθμων για τα διαφορετικά πλήθη δεδομένων εισόδου. Παρατηρούμε, ότι ο πρώτος αλγόριθμος αναζήτησης είναι αρκετά πιο γρήγορος από τον δεύτερο. Σύμφωνα όμως με την θεωρητική ανάλυση για την εκτέλεση του πρώτου αλγορίθμου αναζήτησης χρειάζεται χρόνος $O(\log n/P)$, ενώ ο δεύτερος χρειάζεται χρόνο $O(\log n / \log P)$. Δηλαδή, ασυμπτωτικά ο πρώτος αλγόριθμος παράλληλης αναζήτησης

είναι πιο αργός από τον δεύτερο. Τα πιο πάνω αποτελέσματα όμως δείχνουν το αντίθετο. Αυτό οφείλεται στο ότι ο πρώτος αλγόριθμος υλοποιείται με πιο απλό τρόπο σε σχέση με τον δεύτερο. Συγκεκριμένα, υπάρχουν λιγότερες και πιο απλές πράξεις στο εσωτερικό της επανάληψης του αλγορίθμου, οι οποίες αν και δεν επηρεάζουν τον ασυμπτωτικό χρόνο, αυξάνουν τον πραγματικό χρόνο. Επιπλέον, ο πρώτος αλγόριθμος έχει λιγότερες ταυτόχρονες αναγνώσεις της ίδιας κυψελίδας της κοινόχρηστης μνήμης. Αυτό έχει ως αποτέλεσμα να σπαταλείται αρκετά λιγότερος χρόνος για πρόσβαση στην κοινόχρηστη μνήμη. Από την Γραφική 7.6, στην οποία φαίνονται τα αποτελέσματα για της εκτέλεσης του πρώτου αλγορίθμου με οχτώ επεξεργαστές και του δεύτερου με εφτά, μπορούμε να παρατηρήσουμε ακόμα κάτι που επηρεάζει τα αποτελέσματα. Όπως συμπεράναμε προηγουμένως, στην περίπτωση που ένας αλγόριθμος εκτελείται με μεγαλύτερο αριθμό επεξεργαστών είναι γρηγορότερος σε σχέση με την εκτέλεση του με τη χρήση λιγότερων επεξεργαστών. Το γεγονός αυτό όμως δεν επηρεάζει ιδιαίτερα τις συγκρίσεις αφού ο αριθμός αυτός διαφέρει μόνο κατά ένα επεξεργαστή. Έτσι, η διαφορά στον χρόνο οφείλεται κυρίως στους παράγοντες που αναφερθήκαν προηγουμένως στην παράγραφο αυτή.



Γραφική 7.7 – Σύγκριση Αλγορίθμων Παράλληλης Αναζήτησης

Για θέματα κόστους, αναμένουμε με βάση την θεωρητική ανάλυση των αλγορίθμων ότι το ο δεύτερος αλγόριθμος είναι πιο αποδοτικός σε θέμα πολυπλοκότητας κόστους. Αντίθετα όμως, όπως παρατηρούμε και από την πιο πάνω γραφική παράσταση, το κόστος του πρώτου αλγορίθμου είναι μικρότερο από το κόστος του δεύτερου. Αν και το θεωρητικό κόστος του πρώτου αλγορίθμου είναι μεγαλύτερο, το αποτέλεσμα αυτό θεωρείται λογικό. Αφού το κόστος και των δύο αλγόριθμων εξαρτάτε από τον αριθμό των επεξεργαστών και τον χρόνο εκτέλεσης και ο χρόνος εκτέλεσης του πρώτου αλγορίθμου είναι μικρότερος, τότε δικαιολογημένα παίρνουμε αυτά τα αποτελέσματα.

Από την πιο πάνω σύγκριση των αλγορίθμων, μπορούμε να συμπεράνουμε ότι αν και βάσει της θεωρητικής ανάλυσης ο δεύτερος αλγόριθμος παράλληλης αναζήτησης είναι ο καλύτερος, ωστόσο βάσει της πειραματικής αξιολόγησης ο πρώτος αλγόριθμος είναι πιο αποδοτικός όσο αφορά τον χρόνο, αλλά και το κόστος εκτέλεσης.

Κεφάλαιο 8

Συμπεράσματα

8.1 Τελικά Συμπεράσματα

Το κύριο μέρος της διπλωματικής αυτής εργασίας, ήταν η υλοποίηση, η προσομοίωση και η πειραματική αξιολόγηση παράλληλων αλγορίθμων. Από την πειραματική αξιολόγηση που έγινε, πάρθηκαν κάποια συμπεράσματα μέσω πινάκων μετρήσεων και γραφικών παραστάσεων.

Για όλους τους αλγόριθμους, παρατηρήθηκε η σχέση μεταξύ του πλήθους των δεδομένων και του χρόνου εκτέλεσης του αλγορίθμου. Από τις μετρήσεις που λήφθηκαν, συμπεραίνουμε ότι σε όλους τους αλγόριθμους ο χρόνος εκτέλεσης εξαρτάται από το πλήθος των δεδομένων εισόδου. Συγκεκριμένα, όταν το πλήθος των δεδομένων αυτών αυξάνεται, τότε και ο χρόνος εκτέλεσης αυξάνεται.

Στην περίπτωση των αλγορίθμων παράλληλης αναζήτησης επιβεβαιώθηκε η θεωρητική ανάλυση, βάσει της οποίας ο χρόνος εκτέλεσης εκτός από το πλήθος των δεδομένων εισόδου, εξαρτάται και από των αριθμό των επεξεργαστών που χρησιμοποιούνται κατά την εκτέλεση του αλγορίθμου. Συγκεκριμένα, παρατηρήθηκε ότι διατηρώντας σταθερό το πλήθος των δεδομένων εισόδου και αυξάνοντας το πλήθος των επεξεργαστών που επιλύουν το πρόβλημα της αναζήτησης, ο χρόνος εκτέλεσης μειώνεται λογαριθμικά.

Για τον κάθε αλγόριθμο, έγινε η σύγκριση πρακτικής και θεωρητικής πολυπλοκότητας του χρόνου και του κόστους. Από τις συγκρίσεις αυτές συμπεραίνουμε ότι η θεωρητική πολυπλοκότητα ταυτίζεται ασυμπτωτικά με την πραγματική για μεγάλο πλήθος δεδομένων εισόδου. Υπάρχουν κάποιες διαφορές βέβαια μεταξύ των δύο, οι οποίες είναι κάποιες σταθερές τιμές.

Κατά την αξιολόγηση έγινε σύγκριση των δύο μη βέλτιστων αλγορίθμων προθεματικού αθροίσματος, οι οποίοι έχουν ίδιο θεωρητικό χρόνο αναζήτησης αλλά ο ένας αλγόριθμος υλοποιήθηκε στο μοντέλο EREW PRAM και ο άλλος στο CREW PRAM. Από τις μετρήσεις συμπεράναμε ότι ο αλγόριθμος που είναι υλοποιημένος στο μοντέλο EREW PRAM εκτελείται πιο γρήγορα σε σχέση με τον άλλο αλγόριθμο. Σημαντικός λόγος για αυτό το γεγονός είναι η ταυτόχρονη πρόσβαση στη κοινόχρηστη μνήμη που έχει περισσότερο κόστος χρόνου.

Στη περίπτωση που έγινε η υλοποίηση μη βέλτιστου και βέλτιστου αλγόριθμου για την επίλυση ενός παράλληλου προβλήματος, έγινε η σύγκριση μεταξύ του χρόνου και του κόστους εκτέλεσης των αλγορίθμων. Σε όλες τις περιπτώσεις καταλήξαμε στο ότι όντως ο βέλτιστος αλγόριθμος έχει μικρότερο κόστος όπως αναμενόταν και βάσει της θεωρητικής αξιολόγησης. Στην περίπτωση του χρόνου, όπου μη βέλτιστος και βέλτιστος αλγόριθμος έχουν την ίδια θεωρητική πολυπλοκότητα, υπήρξαν μη αναμενόμενα αποτελέσματα. Ο χρόνος εκτέλεσης του βέλτιστου αλγορίθμου ήταν μεγαλύτερος από τον μη βέλτιστο. Η διαφορά αυτή όμως είναι μια σταθερά η οποία παραμένει σταθερή ή μεταβάλλεται ελαφρώς με την αύξηση του πλήθους των δεδομένων εισόδου.

Συνοψίζοντας, με την αξιολόγηση των αλγορίθμων κατανοήθηκε καλύτερα η πρακτική συμπεριφορά τους σε σχέση με την απόδοση, αλλά επιβεβαιώθηκε και η θεωρητική αξιολόγηση.

8.2 Προβλήματα Υλοποίησης

Τα προβλήματα που παρουσιάστηκαν κατά την εκπόνηση της διπλωματικής εργασίας αφορούσαν κυρίως την υλοποίηση των αλγορίθμων στη γλώσσα XMTC. Εξαιτίας του

ότι η γλώσσα και ο μεταγλωττιστής της δεν είναι πλήρως ανεπτυγμένοι, παρουσιάζονται κάποιοι περιορισμοί που αφορούν την υλοποίηση των αλγορίθμων.

Αρχικά, είναι αδύνατη η κλήση οποιουδήποτε είδους συνάρτησης μέσα από το παράλληλο κομμάτι του κώδικα. Αυτό, είχε ως αποτέλεσμα την αδυναμία υλοποίησης αλγορίθμων που περιέχουν αναδρομή, όπως του αλγορίθμου συγχώνευσης. Ο αλγόριθμος αυτός χρησιμοποιεί αναδρομικά τον δεύτερο αλγόριθμο αναζήτησης που περιγράφτηκε στο Κεφάλαιο 7 για τη συγχώνευση δύο ταξινομημένων λιστών, σε μία επίσης ταξινομημένη λίστα [4, 5]. Άρα, είναι αδύνατο να υλοποιηθεί προς το παρόν στη γλώσσα XMTC. Επιπλέον, έπρεπε να γίνει ο υπολογισμός λογαρίθμων και ρίζας ενός αριθμού, αφού η κλήση των αντίστοιχων συναρτήσεων δεν επιτρέπεται, ούτε εκτός του σειριακού κομματιού του κώδικα.

Ένας άλλο πρόβλημα είναι ο περιορισμός της γλώσσας ότι κάθε παράλληλο κομμάτι του κώδικα δεν πρέπει να ξεπερνά ένα συγκεκριμένο αριθμό εντολών. Στην περίπτωση που κάτι τέτοιο συνέβαινε, έπρεπε η λύση του προβλήματος να διασπαστεί σε περισσότερα κομμάτια παράλληλου κώδικα. Κάποιες φορές όμως ακόμα και η διάσπαση αυτή δεν ήταν αρκετή. Ως αποτέλεσμα, είναι προς το παρόν αδύνατη η ανάπτυξη πολύπλοκων παράλληλων αλγορίθμων στη γλώσσα αυτή.

Στη συνέχεια, είναι αδύνατη η εισαγωγή δεδομένων εισόδου από τον χρήστη του προγράμματος κατά την προσομοίωση του. Άρα, τα δεδομένα πρέπει να δίνονται πάντα εκ των προτέρων, κάτι που σε κάποιες περιπτώσεις δεν ευνοεί την ρεαλιστική αναπαράσταση των προβλημάτων.

Τέλος, υπάρχει περιορισμός στον αριθμό των επεξεργαστών που μπορούν να χρησιμοποιηθούν για την εκτέλεση των πράξεων. Συγκεκριμένα, με τη χρήση του προσομοιωτή δεν μπορούμε να χρησιμοποιούμε αριθμό επεξεργαστών μεγαλύτερο από 1024. Το γεγονός αυτό έχει ως αποτέλεσμα να υπάρχει άνω φράγμα και στο πλήθος των δεδομένων εισόδου του προβλήματος, αφού σε πολλές περιπτώσεις ο αριθμός των επεξεργαστών που χρησιμοποιούνται για την επίλυση ενός προβλήματος καθορίζεται βάσει του πλήθους των δεδομένων εισόδου. Σε αρκετές περιπτώσεις μετρήσεων που λήφθηκαν από την προσομοίωση των αλγορίθμων, παρατηρήθηκε ότι ακόμα κι αν ο

αριθμός των επεξεργαστών δεν περιορίζει ένα συγκεκριμένο πλήθος δεδομένων, ο αλγόριθμος δεν μπορεί να προσομοιωθεί δίνοντας ορθά αποτελέσματα. Αυτό συμβαίνει εξαιτίας των περιορισμών μνήμης που υπάρχουν. Συγκεκριμένα, για μεγάλο αριθμό επεξεργαστών και δεδομένων εισόδου, ιδιαίτερα για αλγόριθμους με σχετικά πολύπλοκες πράξεις, ξεπερνιούνται οι δυνατότητες μνήμης.

8.3 Οφέλη Διπλωματικής Εργασίας

Τα οφέλη από την εκπόνηση της διπλωματικής εργασίας ήταν πολλά. Αρχικά, είχα την ευκαιρία να κατανοήσω την σωστή μεθοδολογία για την ολοκλήρωση μιας εργασίας, σε σχέση με την μελέτη της βιβλιογραφίας, την εκμάθηση της γλώσσας προγραμματισμού XMTC, καθώς και την ανάπτυξη και αξιολόγηση ενός παράλληλου αλγόριθμου.
Σημαντικό είναι το γεγονός ότι μπόρεσα να κατανοήσω πρακτικά τον τρόπο ανάπτυξης ενός παράλληλου αλγόριθμου, και συγκεκριμένα των παράλληλων αλγορίθμων που μέχρι τώρα γνώριζα μόνο σε θεωρητικό στάδιο. Δηλαδή, μέσω της υλοποίησης των αλγορίθμων κατανόησα καλύτερα τους περιορισμούς τόσο του παραλληλισμού, όσο και της γλώσσας προγραμματισμού. Το χρήσιμο μέρος αυτού ήταν ότι αναγκάστηκα να βρω εναλλακτικές λύσεις στα προβλήματα που προκλήθηκαν από τους περιορισμούς και να αναπτύξω τον τρόπο σκέψης μου. Έπειτα, η πειραματική αξιολόγηση των αλγορίθμων βοήθησε στην ανάπτυξη κριτικής σκέψης και εξαγωγής συμπερασμάτων.

Συνοψίζοντας, μέσα από όλη την μελέτη που έγινε κατά την διάρκεια της ανάπτυξης της διπλωματικής εργασίας, κατανοήθηκε καλύτερα και σε περισσότερο βάθος η σημαντικότητα του παράλληλου υπολογισμού. Τέλος, έγινε καλύτερη εκμάθηση των πλεονεκτημάτων που μας παρέχει ο παραλληλισμός στην επίλυση προβλημάτων σε σχέση με την απόδοση και την αποτελεσματικότητα.

8.4 Μελλοντική Εργασία

Για αυτή τη διπλωματική εργασία, μελετήθηκαν και υλοποιήθηκαν κάποιοι παράλληλοι αλγόριθμοι όπου ο καθένας επιλύει ένα συγκεκριμένο πρόβλημα. Για μελλοντική εργασία θα μπορούσαν να υλοποιηθούν πιο πολύπλοκοι αλγόριθμοι, βάσει των οποίων να δοκιμάζονται καλύτερα τα όρια και οι δυνατότητες της γλώσσας XMTC, καθώς και

του προσομοιωτή XMT. Επιπλέον, οι αλγόριθμοι που υλοποιήθηκαν στα πλαίσια αυτής της εργασίας μπορούν να χρησιμοποιηθούν ως μέρος κάποιων άλλων αλγορίθμων πιο πολύπλοκων αλγορίθμων. Απαραίτητη ίσως προϋπόθεση για τα πιο πάνω είναι να υπάρχει μια ολοκληρωμένη ή βελτιωμένη μορφή της γλώσσας προγραμματισμού XMTC και του μεταγλωττιστή της. Τέλος, θα μπορούσαν να υλοποιηθούν οι ίδιοι αλγόριθμοι που υλοποιήθηκαν σε αυτή την εργασία, στη πρωτότυπη μηχανή FPGA. Έτσι, είναι δυνατό να συγκριθεί η πειραματική πολυπλοκότητα των ίδιων αλγορίθμων και να διαπιστωθεί κατά πόσο ο χρόνος προσομοίωσης των αλγορίθμων είναι κοντά στο χρόνο εκτέλεσης τους στην μηχανή αυτή.

Βιβλιογραφία

[1] Aydin O. Balkan, Uzi Vishkin, George C. Caragea, Alexandros Tzannes, "Tutorial for XMTC Language, XMTC Compiler and XMT Simulator", 9 February 2009, http://www.umiacs.umd.edu/users/vishkin/XMT/index.shtml.

[2] Aydin O. Balkan, Uzi Vishkin, George C. Caragea, Alexandros Tzannes, "XMT Toolchain Manual for XMTC Language, XMTC Compiler, XMT Simulator and Paraleap XMT FPGA Computer", 1 January 2009, http://www.umiacs.umd.edu/users/vishkin/XMT/index.shtml.

[3] Lorin Hochstein, Victor R. Basili, Uzi Vishkin, John Gilbert, "A pilot study to compare programming models", 25 December 2007.

[4] Joseph JaJa, "An Introduction to Parallel Algoritms", Addison-Wesley, 1992.

[5] Χρύσης Γεωργίου, Σημειώσεις Μαθήματος ΕΠΛ431 – Σύνθεση Παράλληλων Αλγορίθμων, Τμήμα Πληροφορικής, Πανεπιστήμιο Κύπρου, http://www2.cs.ucy.ac.cy/~chryssis/EPL431/.

[6] Xingzhi Wen, Uzi Vishkin, "FGPA-Based Prototype of a PRAM-On-Chip Processor", 2008.

[7] University of Illinois, "Using Simple Abstraction to Guide the Reinvention of Computing for Parallelism", http://illinois.edu/calendar/Calendar.

[8] Τεχνικές Παράλληλου Προγραμματισμού, http://www.it.uom.gr/project/parallel/

Παράρτημα Α

Κώδικας Αλγορίθμων που Υλοποιήθηκαν

Α.1 Αλγόριθμος Παράλληλης Άθροισης

```
#include <xmtc.h>
#include "sum.h"

// Koinoxristi Metavliti gia tin opoia desmeftike mnimi mesw tou ergaleiou mnimis
// kai enswmatwnetai mesw tou header file sum.h: int A[N]
// A[N] - pinakas ston opoio tha metadwthei enas arithmos

int main()
{
        int end_thread=N/2;

        //Parallel Mode
        spawn(1, end_thread)
        {
                //Topikes Metavlites
                int temp=0;
                int k=2;

                while($<=N/k)
                {
                        A[$]=A[2*$-1]+A[2*$];

                        //Gia prosthiki tn stoixeiwn ta opoia den simperilamvanontai
                        //stous ipologismous (otan to N/k einai monos arithmos opou den
                        //iparxei epexergastis gia na ipologisei to athroisma tou teleftaiou
                        //stoixeiou me arithmo N/k me tous ipoloipous arithmous)
                        temp=N/k;

                        if($==temp && (temp % 2 != 0) && $!=1)
                                A[1]=A[1]+A[$];

                        k=k*2;
                }
```

```
		//Gia prosthiki tou teleftaiou stoixeiou stin periptwsi pou
		// o aritmos tn stoixeiwn einai perittos
		if($==1 && N % 2 != 0)
			A[1]=A[1]+A[N];
	}
}
```

Α.2 Βέλτιστος Αλγόριθμος Παράλληλης Άθροισης

```
#include <xmtc.h>
#include "sum.h"
#define N 512

// Koinoxristi Metavliti gia tin opoia desmeftike mnimi mesw tou ergaleiou mnimis
// kai enswmatwnetai mesw tou header file sum.h: int A[N]
// A[N] - pinakas ston opoio tha metadwthei enas arithmos

int main()
{
	//Koinoxristes Metavlites
	int logn;
	int counter=0;
	int temp=1;
	int max=0;

	//Ypologismos tou logarithmou
	while(N>temp)
	{
		temp=temp*2;
		counter=counter+1;
	}
	logn=counter;

	//Gia na min afinoume stoixeia ektos ipologismou
	max=N/logn;
	if(logn*max!=N)
		max=max+1;
```

```
//Parallel Mode 1
//Seiriaki athroisi stoixeiwn mias ipolistas
spawn(0,max-1)
{
	int k;
	int j;
	int sum;

	int temp2=1;
	int counter2=0;
	int logn2;

	//Ypologismos tou logarithmou
	while(N>temp2)
	{
		temp2=temp2*2;
		counter2=counter2+1;
	}
	logn2=counter2;

	k=1;
	sum=0;
	j=$*logn2;

	//Seiriaki Athroisi twn dedomenwn eisodou
	while(N>k)
	{
		sum=sum+A[j];
		j=j+1;
		k=k*2;
	}
	A[$]=sum;
}

//Parallel Mode 2
//Mi veltistos algorithmos parallilis athroisis
spawn(1,max/2)
{
	int k0=2;
	int tmp=0;

	int temp2=1;
	int counter2=0;
```

```
		int logn2;
		int max2;

		//Ypologismos tou logarithmou
		while(N>temp2)
		{
			temp2=temp2*2;
			counter2=counter2+1;
		}
		logn2=counter2;

		//Gia na min afinoume stoixeia ektos ipologismou
		max2=N/logn2;
		if(logn2*max2!=N)
			max2=max2+1;

		//Parallili Athroisi twn stoixeiwn pou ipologistikan me ti seiriaki athroisi

		while($<=max2/k0)
		{
			A[$-1]=A[2*($-1)]+A[2*($-1)+1];

			tmp=max2/k0;
			if(($==tmp) && (tmp % 2 != 0) && $!=1)
				A[0]=A[0]+A[tmp-1];

			k0=k0*2;
		}

		if($==0 && max2 % 2 != 0)
			A[0]=A[0]+A[max2-1];
	}

}
```

Α.3 Αλγόριθμος Παράλληλης Μετάδοσης

```
#include <xmtc.h>
#include "broadcast.h"
#define N 512

// Koinoxristes Metavlites gia tis opoies desmeftike mnimi mesw tou ergaleiou mnimis
// kai enswmatwnontai mesw tou header file broadcast.h: int n, int A[n]
// n - plithos dedomenwn eisodou
// A[n] - pinakas ston opoio tha metadwthei enas arithmos

int main()
{
        A[1]=1;

        //Parallel Mode
        spawn(1,N/2)
        {
                int k=1;

                //Metadwsi tou aritmou
                while(k<N)
                {
                        if($<=k)
                                A[$+k]=A[$];
                        k=k*2;
                }
        }

        printf("%d elements have been broadcasted\n",N);

}
```

Α.4 Παράλληλος Αλγόριθμος Προθεματικού Αθροίσματος EREW PRAM

```
#include <xmtc.h>
#include "sum.h"

// Koinoxristes Metavlites gia tis opoies desmeftike mnimi mesw tou ergaleiou mnimis
// kai enswmatwnontai mesw tou header file sum.h: int n, int A[n]
// n - plithos dedomenwn eisodou
// A[n] - pinakas me ta dedomena eisodou

int main()
{
        //Parallel Mode
        spawn(1, n-1)
        {
                //Topiki Metavliti
                int s=1;

                //Ypologismos Prothematikou Athroismatos
                while(s<=$)
                {
                        A[$]=A[$-s]+A[$];
                        s=s*2;
                }
        }
}
```

Α.5 Παράλληλος Αλγόριθμος Προθεματικού Αθροίσματος CREW PRAM

```
#include <xmtc.h>
#include "sum.h"

// Koinoxristes Metavlites gia tis opoies desmeftike mnimi mesw tou ergaleiou mnimis
// kai enswmatwnontai mesw tou header file sum.h: int n, int A[n]
// n - plithos dedomenwn eisodou
// A[n] - pinakas me ta dedomena eisodou
```

```
int main()
{

	//Parallel Mode
	spawn(1,n/2)
	{
		//Topikes Metavlites
		int position;
		int s;
		int el;
		int k;

		//Arxikopoiisi Topikwn Metavlitwn
		position=$*2;
		el=position;
		s=1;
		k=1;

		//Ypologismos tou prwtou prothematos tou kathe epexergasti
		A[position]=A[position]+A[position-s];

		while(n>2*k)
		{

			//Ypologismos tis thesis stin opoia tha
			//apothikeftei to epomeno prothema
			el=el/2;
			if(el % 2!=0)
				el=el+1;

			position=(el*k)+$;
			s=$-(el*k)+(2*k);

			//Ypologismos tou prothematos
			A[position]=A[position]+A[position-s];

			k=k*2;
		}
	}
}
```

Α.6 Βέλτιστος Αλγόριθμος Προθεματικού Αθροίσματος EREW PRAM

```
#include <xmtc.h>
#include "sum.h"
#define N 512

// Koinoxristi Metavliti gia tin opoia desmeftike mnimi mesw tou ergaleiou mnimis
// kai enswmatwnonetai mesw tou header file sum.h: int A[n]
// A[n] - pinakas me ta dedomena eisodou

int main()
{
	//Dilwsi Koinoxristwn Metavlitwn
	int temp;
	int counter;
	int logn;
	int max;

	//Ypologismos tou logn
	temp=1;
	counter=0;
	while(N>temp)
	{
		temp=temp*2;
		counter=counter+1;
	}
	logn=counter;

	//Ypologismos megistou arithmou epexergastwn pou tha xrisimopoiithoun
	max=N/logn;
	if(logn*max!=N)
		max=max+1;


	//Parallel mode - seiriako prothematiko athroisma
	spawn(0,max-1)
	{
		//Dilwsi Topikwn Metavlitwn
		int k;
		int j;

		int temp2=1;
```

```
	int counter2=0;
	int logn2;

	//Ypologismos tou logn
	while(N>temp2)
	{
		temp2=temp2*2;
		counter2=counter2+1;
	}
	logn2=counter2;

	//Arxikopoiisi Topikwn Metavlitwn
	k=2;
	j=$*logn2+1;

	//Ypologismos seiriakou athroismatos
	while(N>k)
	{
		A[j]=A[j]+A[j-1];
		j=j+1;
		k=k*2;
	}
}

//Parallel Mode
//Parallilos algorithmos prothematikou athroismatos sto montello EREW
spawn(2,max)
{
	//Dilwsi Topikwn Metavlitwn
	int s;
	int j;

	int temp2=1;
	int counter2=0;
	int logn2;

	//Ypologismos tou logn
	while(N>temp2)
	{
		temp2=temp2*2;
		counter2=counter2+1;
	}
	logn2=counter2;
```

```
	//Arxikopoiisi Topikwn Metavlitwn
	s=logn2;
	j=$*logn2-1;

	//Ypologismos prothematwn
	while(s<=($*logn2))
	{
		A[j]=A[j]+A[j-s];
		s=s*2;
	}
}

//Parallel Mode - seiriako athroisma twn stoixeiwn tou kathe block
spawn(1,max-1)
{
	//Dilwsi Topikwn Metavlitwn
	int k;
	int j;
	int p;

	int temp2=1;
	int counter2=0;
	int logn2;

	//Ypologismos tou logn
	while(N>temp2)
	{
		temp2=temp2*2;
		counter2=counter2+1;
	}
	logn2=counter2;

	//Arxikopoiisi Topikwn Metavlitwn
	k=2;
	p=$*logn2;
	j=$*logn2-1;
```

```
                //Ypologismos athroismatos
                while(N>k)
                {
                        A[p]=A[p]+A[j];
                        p=p+1;
                        k=k*2;
                }
        }
}
```

Α.7 Πρώτος Παράλληλος Αλγόριθμος Αναζήτησης σε Ταξινομημένη Λίστα

```
#include <xmtc.h>
#include "search.h"

// Koinoxristes Metavlites gia tis opoies desmeftike mnimi mesw tou ergaleiou mnimis
// kai enswmatwnontai mesw tou header file search.h: int n, int A[n], int p, int y
// n - plithos dedomenwn eisodou
// A[n] - lista me ta dedomena eisodou
// p - plithos epexergastwn
// y - stoixeio gia anazitisi sti lista A

int main()
{
        int found=0;              //found=1 an to y vrethike sti lista
        int positionOfY=-1;     //i teliki thesi tou y sti lista A

        //Parallel Mode
        spawn(1,p)
        {
                int counter=0;
                int left;          //i arxi tou $ search block
                int right;         //to telos $ search block
                int size;          //to size tou $ search block
                int middle; //thesi tou mesaiou stoixeiou tis parousas ipolistas tou $ block
                int position;     //prosorini thesi tou y

                //Ypologismos twn akrwn k tou megethous tis ipolistas tou epexergasti
                left=($-1)*(n/p)+1;
                right=$*(n/p);
                size=right-left+1;
```

```
//Anazitisi stoixeiou anazitisis y sti lista A
while(size>=1 && found==0 && counter<n/p && right>=left)
{
	counter++;

	middle=left+size/2;
	if(size % 2!=0 && size!=1)
		middle=middle+1;

	if(A[middle]==y)
	{
		positionOfY=middle;
		found=1;
		break;
	}
	else if(A[middle]>y)
	{
		position=middle-1;
		right=middle-1;
	}
	else if(A[middle]<y)
	{
		position=middle;
		left=middle+1;
	}

	size=right-left+1;
}//telos epanalipsis anazitisis


if(A[n/p+1]==(n/p+1) && $!=p)
	positionOfY=position+1;

//Kathorismos tis thesis tou y ean den iparxei sti lista A
//  stoixeio iso me to y
if(found!=1)
{
	//Ean to y einai anamesa sto dexi stoixeio tis $ ipolistas
	//  kai to aristero stoixeio tis $+1 ipolistas
	if($!=p && y>=A[$*(n/p)] && y<=A[$*(n/p)+1])
		positionOfY=position;
```

```
                        //Ean to position einai anamesa sto aristero kai
                        //  dexi stoixeio tis ipolistas
                        else if(position>($-1)*(n/p)+1 && position<$*(n/p))
                                positionOfY=position;

                        //Ean to position einai iso me tin aristeri thesi tis $ ipolistas
                        else if(position==($-1)*(n/p)+1)
                                positionOfY=position;

                        //Ean to position einai ektos twn oriwn tis listas A
                        else if(position<=1)
                                positionOfY=position;
                        else if(position>=n-1)
                                positionOfY=n;
                }

        } //telos tou parallel mode

        printf("Position of %d in A = %d\n",y,positionOfY);
}
```

Α.8 Δεύτερος Παράλληλος Αλγόριθμος Αναζήτησης σε Ταξινομημένη Λίστα

```
#include <xmtc.h>
#include "search.h"

// Koinoxristes Metavlites gia tis opoies desmeftike mnimi mesw tou ergaleiou mnimis
// kai enswmatwnontai mesw tou header file search.h: int n, int x[n+1], int p, int y
// n - plithos dedomenwn eisodou
// x[n+1] - lista me ta dedomena eisodou taxinomimena
// p - plithos epexergastwn
// y - stoixeio gia anazitisi sti lista x

int main()
{
        int found=0;
        int positionOfY=0;      //teliki thesi tou y
        int id[p+1];
        int C[p+1];
        int l;                  //arxi tis listas
        int r;                  //telos tis listas
```

```
spawn(1,p)
{
	//Topikes Metavlites
	int subEl=0;
	int temp=0;
	int j=0;

	//Arxikipoiisi Koinoxristwn Metavlitwn
	if($==1)
	{
		C[0]=0;
		C[p+1]=1;
		l=0;
		r=n+1;
	}

	//Anazitisi stoixeiou y sti lista x
	while((r-l>p) && found==0 && j<p)
	{
		j=j+1;

		if($==1)
		{
		id[0]=l;
		id[p+1]=r;
		}

		//Ypologismos akraiwn thesewn tis parousas ipolistas
		subEl=(r-l)/(p+1);
		id[$]=l+$*subEl;

		temp=id[$];

		if(y==x[temp])
		{
			positionOfY=temp;
			found=1;
		}
		else if(y>x[temp])
			C[$]=0;
		else if(y<x[temp])
			C[$]=1;
```

```
                if(C[$]<C[$+1])
                {
                        l=id[$];
                        r=id[$+1];
                }

                if($==1 && C[0]<C[1])
                {
                        l=id[0];
                        r=id[1];
                }
        }//end of while

        //Ypologismos thesis tou y ean den vrethike sti lista x
        if(found==0 && $<=r-l)
        {
                temp=l+$;
                if(y==x[temp])
                {
                        positionOfY=temp;
                        found=1;
                }
                else if(y>x[temp])
                        C[$]=0;
                else if(y<x[temp])
                        C[$]=1;


                if(C[$-1]<C[$])
                        positionOfY=l+$-1;
        } //end if

        if(y>x[n])
                positionOfY=n;
        else if(y<x[1])
                positionOfY=0;

    } //end of parallel mode

    printf("The position of %d is",y);
    printf(" %d\t\n\n",positionOfY);

} //end of program
```