Ατομική Διπλωματική Εργασία

ΠΡΟΣΟΜΟΙΩΣΗ ΚΑΙ ΠΕΙΡΑΜΑΤΙΚΗ ΑΞΙΟΛΟΓΗΣΗ ΕΥΡΩΣΤΩΝ ΠΑΡΑΛΛΗΛΩΝ ΑΛΓΟΡΙΘΜΩΝ ΣΤΗΝ ΠΛΑΤΦΟΡΜΑ XMT

Ευστάθιος Ιωακείμ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2012

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΠΡΟΣΟΜΟΙΩΣΗ ΚΑΙ ΠΕΙΡΑΜΑΤΙΚΗ ΑΞΙΟΛΟΓΗΣΗ ΕΥΡΩΣΤΩΝ ΠΑΡΑΛΛΗΛΩΝ ΑΛΓΟΡΙΘΜΩΝ ΣΤΗΝ ΠΛΑΤΦΟΡΜΑ XMT

Ευστάθιος Ιωακείμ

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 2012

Ευχαριστίες

Θα ήθελα να εκφράσω τις ευχαριστίες μου στον επιβλέπων καθηγητή μου, κύριο Χρύση Γεωργίου, για την καθοδήγηση και βοήθεια που μου προσέφερε καθ' όλη την διάρκεια ολοκλήρωσης της παρούσας Ατομικής Διπλωματικής Εργασίας.

Περίληψη

Η παρούσα Ατομική Διπλωματική Εργασία (ΑΔΕ) ασχολείται με την μελέτη του παραλληλισμού στην επιστήμη της πληροφορικής. Ο όρος παραλληλισμός αναφέρεται στην ταυτόχρονη επεξεργασία δεδομένων από πολλαπλούς επεξεργαστές με σκοπό την εξαγωγή της επιθυμητής πληροφορίας.

Πιο συγκεκριμένα στην παρούσα ΑΔΕ γίνεται υλοποίηση και προσομοίωση παράλληλων αλγορίθμων στην πλατφόρμα XMT, με σκοπό την σύγκριση των αποτελεσμάτων – μετρήσεων που προκύπτουν με τη θεωρητική ανάλυση των αντίστοιχων παράλληλων αλγορίθμων του μοντέλου PRAM και A-PRAM. Η πλατφόρμα XMT μας δίδει την δυνατότητα υλοποίησης των παράλληλων αλγορίθμων των μοντέλων PRAM και A-PRAM αλλά επίσης μας δίδει και τα μέσα για να προσομοιώσουμε τους αλγορίθμους σε ένα απλό υπολογιστή.

Κάθε ένας από τους παράλληλους αλγορίθμους που υλοποιήθηκε έχει μελετηθεί ξεχωριστά για την σχέση του με το θεωρητικό του πρότυπο. Επίσης γίνεται αναφορά, στο τελευταίο κεφάλαιο της παρούσας ΑΔΕ, για την συνολική συμπεριφορά της πλατφόρμας XMT έναντι των θεωρητικών μοντέλων PRAM και A-PRAM.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Κίνητρο

Σε μια εποχή όπου ο παραλληλισμός απέδειξε την αξία του και εγκαθιδρύθηκε ως το επόμενο βήμα στην ταχύτερη ολοκλήρωση πάσης φύσεως εργασιών· χρίζει μελέτης προκειμένου να εκμεταλλευτούμε πλήρως τις δυνατότητες που μας παρέχει. Χαρακτηριστικά να πούμε ότι η εμφάνιση των πολυπύρηνων επεξεργαστών αλλά και η ανάγκη για την επίλυση πιο δύσκολων και μεγαλύτερων προβλημάτων ήταν τα σημεία κλειδιά τα οποία καθιέρωσαν τον παραλληλισμό ως το επόμενο βήμα στην εξέλιξη της επιστήμης της Πληροφορικής. Συνεπώς υπάρχει ανάγκη για προγραμματισμό των πολυπυρήνων.

Το θεωρητικό μοντέλο PRAM, μελετείται εδώ και δεκαετίες και βοηθά στην παράλληλη σκέψη, αλλά μέχρι πρόσφατα δεν ήταν εφικτή η εύκολη υλοποίηση της πλειάδας αλγορίθμων που αναπτύχθηκαν στο μοντέλο αυτό. Σήμερα μέσω της πλατφόρμας XMT αυτό είναι εφικτό. Η σχετική απλότητα στον προγραμματισμό της πλατφόρμας, βασική γνώση PRAM και γνώση C, την καθιστά ελκυστική για έρευνα. Είναι μία ευκαιρία για να μελετήσουμε τις δυνατότητες της πλατφόρμας, υλοποιώντας προχωρημένους αλγόριθμους. Επίσης η ενασχόληση μου με την υλοποίηση παράλληλων αλγορίθμων είναι μία ευκαιρία που μου δόθηκε για να εμβαθύνω τις γνώσεις μου στον παράλληλο προγραμματισμό.

1.2 Στόχοι

Η παρούσα Ατομική Διπλωματική Εργασία αποσκοπεί στην υλοποίηση και μελέτη παράλληλων αλγορίθμων στην πλατφόρμα XMT. Με τους όρους υλοποίηση και μελέτη, εννοούμε την εκμάθηση της πλατφόρμας XMT ούτως ώστε να είναι δυνατή η υλοποίηση των αλγορίθμων αλλά και την σύγκριση των αποτελεσμάτων που προκύπτουν από τις προσομοιώσεις με τα θεωρητικά πρότυπα των αλγορίθμων. Επίσης γίνεται υλοποίηση ενός από τους παράλληλους αλγορίθμους (παράλληλης άθροισης) σε διαφορετικές εκδοχές, όπου σε κάθε εκδοχή ακολουθείται διαφορετική χαρτογράφηση της κοινόχρηστης μνήμης (memory mapping). Ο λόγος για τον οποίο γίνεται αυτή η μελέτη της χαρτογράφησης της μνήμης είναι ότι επιθυμούμε να μελετήσουμε το βαθμό που επηρεάζεται το κόστος και ο χρόνος εκτέλεσης του αλγορίθμου στην πλατφόρμα XMT.

Συνοπτικά οι αλγόριθμοι που μελετήθηκαν ήταν ο αλγόριθμος παράλληλης άθροισης και οι αλγόριθμοι W [1] και X [1] που επιλύουν το πρόβλημα Write-All. Το πρόβλημα της παράλληλης άθροισης αφορά την άθροιση *n* στοιχείων αποθηκευμένων στην κοινόχρηστη μνήμη. Το πρόβλημα Write-All αναφέρεται στην ανάγκη, δεδομένου πλήθους κυψελίδων n, αρχικοποιημένες με την τιμή 0 (μηδέν), να αποθηκευτεί σε κάθε κυψελίδα μνήμης η τιμή 1 (ένα). Γίνεται μια σύντομη και γενική αναφορά στους αλγόριθμους και τα προβλήματα που επιλύουν εξαιτίας του ότι γίνεται αναλυτική αναφορά στα μετέπειτα σχετικά κεφάλαια της Ατομικής Διπλωματικής Εργασίας.

Γενικά η μελέτη των προαναφερθέντων παράλληλων αλγορίθμων θα μας δώσει τα εφόδια (μετρήσεις) ούτος ώστε να επιτύχουμε την σύγκριση μεταξύ πρακτικής υλοποίησης των αλγορίθμων και των θεωρητικών τους προτύπων αλλά και να ελέγξουμε τις δυνατότητες της παρούσας έκδοσης της πλατφόρμας XMT.

1.3 Μεθοδολογία

Για την υλοποίηση της Ατομικής Διπλωματικής Εργασίας εργάστηκα ταυτοχρόνως γύρω από δύο βασικούς άξονες με την επίβλεψη και την καθοδήγηση του επιβλέπων καθηγητή μου, κύριου Χρύση Γεωργίου. Ο πρώτος άξονας αφορά την θεωρητική πλευρά του αντικειμένου ενώ ο δεύτερος άξονας αναφέρεται στο πρακτικό μέρος το οποίο σχετίζεται με την πλατφόρμα XMT. Καταρχάς προτού προχωρήσω σε αναλυτικότερη επεξήγηση της μεθοδολογίας που ακολούθησα· αξίζει να αναφέρω ότι η

επαφή μου με το αντικείμενο των παράλληλων αλγορίθμων έχει ξεκινήσει από πιο παλιά, με την παρακολούθηση του μαθήματος "ΕΠΛ431 Σύνθεση Παράλληλων Αλγορίθμων". Το μάθημα αυτό έθεσε τα θεμέλια των γνώσεων μου σχετικά με τους παράλληλους αλγόριθμους. Πιο συγκεκριμένα, στο μάθημα αυτό έγινε αναλυτική αναφορά στον αλγόριθμο παράλληλης άθροισης καθώς επίσης και αναφορά στους αλγορίθμους W και X. Το κυρίως μοντέλο παράλληλου προγραμματισμού που χρησιμοποιήθηκε στο μάθημα ήταν το PRAM ούτως ώστε να θεμελιωθούν οι βασικές γνώσεις παράλληλης αλγοριθμικής σκέψης. Επιπλέον είχε γίνει αναφορά στο μοντέλο A-PRAM.

Αρχικά με τις κατευθυντήριες οδηγίες του επιβλέπων καθηγητή μου, ξεκίνησα την εις βάθος μελέτη των παράλληλων αλγορίθμων που θα υλοποιούσα. Επίσης, βάσει των πληροφοριών από τον διαδικτυακό χώρο της πλατφόρμας XMT [2], εγκατέστησα την πλατφόρμα XMT μαζί με το σύνολο των εργαλείων που προαπαιτούνταν στον προσωπικό μου υπολογιστή. Όπως τόνισα και στην αρχή του υποκεφαλαίου· εργαζόμουν παράλληλα στο θεωρητικό αλλά και στο πρακτικό κομμάτι της Ατομικής Διπλωματικής Εργασίας. Στις επόμενες παραγράφους ακολουθεί αναλυτικότερη περιγραφή για τους δύο άξονες εργασίας.

Παρότι είχα γνώση των παράλληλων αλγορίθμων ήταν αναγκαία η μελέτη τους σε βάθος. Συγκεκριμένα ήταν αναγκαία η εις βάθος κατανόηση τους διότι το προγραμματιστικό μοντέλο που παρέχει η πλατφόρμα XMT έχει κάποιες ιδιαιτερότητες (δες κεφάλαιο Κεφάλαιο 3) με επακόλουθο την ανάγκη μερικής προσαρμογής του κώδικα των αλγορίθμων ώστε να επιτύχω την ισοδύναμη αναπαράσταση τους στην πλατφόρμα XMT. Αναλυτικότερα για τους αλγόριθμους που μελετήθηκαν, γίνονται αναφορές στα αντίστοιχα κεφάλαια (Κεφάλαιο 4, Κεφάλαιο 5, Κεφάλαιο 6).

Ο δεύτερος άξονας σχετίζεται με την πλατφόρμα XMT. Η κατανόηση της πλατφόρμας στο σύνολο της ήταν σημαντικό ούτως ώστε να είμαι σε θέση να υλοποιήσω και να προσομοιώσω τους παράλληλους αλγορίθμους. Οι ιδιαιτερότητες της γλώσσας παράλληλου προγραμματισμού XMTC αλλά και η κατανόηση του παράλληλου προγραμματιστικού μοντέλου που υλοποιεί η πλατφόρμα XMT, ήταν δύο σημαντικοί παράμετροι για την επιτυχή διεκπεραίωση της Ατομικής Διπλωματικής Εργασίας. Πιο συγκεκριμένες αναφορές για την πλατφόρμα XMT γίνονται στο Κεφάλαιο 3.

Κατά την διάρκεια εργασίας στους δύο άξονες, γίνονταν και συναντήσεις με τον επιβλέπων καθηγητή, ο οποίος μέσω συμβουλών και υποδείξεων (οι οποίες αρκετές φορές ήταν καθοριστικού χαρακτήρα) με βοήθησε στην ομαλή πρόοδο της εργασίας μου. Επίσης ορισμένες συμβουλές και υποδείξεις είχαν επιμορφωτικό χαρακτήρα. Μετά την επιτυχή υλοποίηση των αλγορίθμων έγινε η αξιολόγηση τους βάση σεναρίων. Συγκεκριμένα λήφθηκαν μετρήσεις για κάθε αλγόριθμο ξεχωριστά και στην συνέχεια έγινε επεξεργασία με την χρήση του προγράμματος Microsoft Excel. Επίσης κατά την πορεία ολοκλήρωσης της Ατομικής Διπλωματικής Εργασίας κρατούσα σημειώσεις και αρχείο για τα οποιαδήποτε σημαντικά αποτελέσματα προέκυπταν ούτως ώστε να τα χρησιμοποιήσω κατά την συγγραφή της ΑΔΕ.

1.4 Δομή Κειμένου

Στο Κεφάλαιο 2 γίνεται αναφορά στον παραλληλισμό, τους παράλληλους αλγόριθμους και τα μοντέλα παράλληλου υπολογισμού. Στο Κεφάλαιο 3 αναλύεται η πλατφόρμα XMT, η ιστορία της, η αρχιτεκτονική της και μια πρώτη επαφή με την γλώσσα προγραμματισμού XMTC. Το Κεφάλαιο 4 ασχολείται με την παράλληλη άθροιση. Πιο συγκεκριμένα παρουσιάζει το πρόβλημα της άθροισης, τον παράλληλο αλγόριθμο που το επιλύει μαζί με τις διαφορετικές υλοποιήσεις του και στο τέλος την αξιολόγηση τους. Στο Κεφάλαιο 5 γίνεται αναφορά στον Αλγόριθμο W. Στην αρχή του κεφαλαίου ορίζεται το πρόβλημα Write-All ενώ στην συνέχεια γίνεται ανάλυση της υλοποίησης και στο τέλος ακολουθεί η αξιολόγηση. Να πούμε ότι σε αυτό το κεφάλαιο γίνεται ανάλυση του μηχανισμού πρόκλησης σφαλμάτων, ο οποίος υλοποιήθηκε προκειμένου να είναι δυνατή η προσομοίωση σεναρίων με σφάλματα. Στο Κεφάλαιο 6 αναλύεται και αξιολογείται ο Αλγόριθμος X. Τελειώνοντας, στο Κεφάλαιο 7 καταγράφονται τα συνολικά συμπεράσματα, προβλήματα και οφέλη από την Ατομική Διπλωματική Εργασία. Επίσης γίνεται αναφορά και σε μελλοντικές εργασίες που μπορούν να γίνουν και σχετίζονται με την πλατφόρμα XMT.

Κεφάλαιο 2

Παράλληλος Υπολογισμός

2.1 Παραλληλισμός

Ο παραλληλισμός ως έννοια ορίζεται σαν η ταυτόχρονη εκτέλεση δύο ή περισσοτέρων διαδικασιών (αναφέρομαι στην έννοια *διαδικασία* με την γενική της μορφή, όχι με τον ορισμό της Επιστήμης της Πληροφορικής) με σκοπό την επίτευξη ενός κοινού στόχου. Καθημερινά όλοι μας πολλές φορές χωρίς να το αντιλαμβανόμαστε λαμβάνουμε μέρος στην εκτέλεση παράλληλων διαδικασιών ή ακόμη εποπτεύουμε τέτοιες διαδικασίες.

Υπάρχει πληθώρα τέτοιων παραδειγμάτων, παράλληλων διαδικασιών, που μπορεί να αναφέρει κανείς· περιλαμβάνει από τις πιο απλές μέχρι τις πιο σύνθετες διαδικασίες. Ένα απλό παράδειγμα είναι η διόρθωση των γραπτών των τελειόφοιτων μαθητών Λυκείου για την εισαγωγή τους στα Ακαδημαϊκά Ιδρύματα. Υπάρχουν αρκετές χιλιάδες γραπτά τα οποία για να διορθωθούν σε εύλογο χρονικό διάστημα απαιτείται η συνδρομή εκατοντάδων καθηγητών. Αξίζει να σημειώσουμε ότι η πιο σημαντική παράμετρος είναι το χρονικό περιθώριο για την ολοκλήρωση της διόρθωσης όλων των γραπτών ούτως ώστε να καταρτιστούν οι λίστες κατάταξης στα Ακαδημαϊκά Ιδρύματα εγκαίρως. Για την επίτευξη του στόχου, όπως προαναφέραμε, γίνεται ταυτόχρονη / παράλληλη διόρθωση των γραπτών από εκατοντάδες καθηγητές. Δεν θα μπούμε σε

περισσότερες λεπτομέρειες για τον τρόπο διόρθωσης των γραπτών αλλά θα τονίσουμε ότι κάθε καθηγητής αναλαμβάνει να διορθώσει (διεκπεραιώσει) ένα προκαθορισμένο μικρό αριθμό γραπτών. Κοιτάζοντας το πρόβλημα από πιο αφαιρετική οπτική γωνία, μόλις ολοκληρώσουν όλοι οι καθηγητές την δουλειά τους τότε ο μηχανισμός – σύστημα είναι σε θέση να ετοιμάσει μια ενιαία λίστα κατάταξης στα Ακαδημαϊκά Ιδρύματα με βάση τις βαθμολογίες των μαθητών. Αυτό ήταν ένα από τα πολλά απλά παραδείγματα παραλληλισμού. Μέθοδοι παραλληλισμού εφαρμόζονται σε αρκετές δουλειές όπως ζαχαροπλαστεία, οικοδομές και άλλες πολλές. Από τα πιο σύνθετα παραδείγματα παραλληλισμού είναι η κατασκευή πολύπλοκων τεχνολογικών επιτευγμάτων από την ανθρωπότητα (διαστημόπλοια, τεχνητοί δορυφόροι, γέφυρες, κ.α.). Χωρίς την ταυτόχρονη / παράλληλη συνεισφορά από οργανισμούς ή ιδιώτες ίσως τα σημερινά επίπεδα γνώσεων και ποιότητας ζωής να ήταν ανύπαρκτα. Ιστορικό σημείο για την εφαρμογή του παραλληλισμού (γενικότερη έννοια) στην βιομηχανία ήταν η περίοδος 1750-1850 (Βιομηχανική Επανάσταση). Κλείνοντας την αναφορά σε γενικά παραδείγματα παραλληλισμού αξίζει να τονίσουμε τα σημαντικά πλεονεκτήματα του έναντι της σειριακής ολοκλήρωσης εργασιών. Ας διαφοροποιήσουμε το σενάριο που μελετήσαμε πιο πριν με τις λίστες κατάταξης στα Ακαδημαϊκά Ιδρύματα. Έστω ότι όλα τα γραπτά πρέπει να διορθωθούν από ένα και μόνο καθηγητή (περίπτωση κατά την οποία γίνεται σειριακή επίλυση). Σαφώς ακούγεται αστείο αλλά δεν παύει να είναι ένα χαρακτηριστικό σενάριο ώστε να τονιστούν τα πλεονεκτήματα της παράλληλης έναντι της σειριακής επίλυσης προβλημάτων. Κυρίως ο μειωμένος χρόνος ολοκλήρωσης σύνθετων εργασιών, γενικότερα, αλλά και η μειωμένη προσπάθεια που απαιτείται από το ευρύτερο σύνολο (σύνολο των ανθρώπων) οδήγησαν στην θεμελίωση - ενσωμάτωση του παραλληλισμού στη ζωή μας.

Η Επιστήμη της Πληροφορικής ως γνωστών ασχολείται με την επίλυση προβλημάτων, παραγωγή πληροφορίας από την επεξεργασία δεδομένων, και γενικότερα να υποβοηθά το έργο των ανθρώπων. Απλώς μια γενική αναφορά για την πορεία του κλάδου· αξίζει να τονίσουμε ότι η αρχική φιλοσοφία για την θεμελίωση του στηριζόταν στον σειριακό τρόπο λειτουργίας και επίλυσης προβλημάτων. Με την πάροδο του χρόνου άρχισε να εμφανίζεται η ανάγκη για ταχύτερους υπολογιστές οι οποίοι να είναι σε θέση να επιλύουν πιο σύνθετα και μεγαλύτερα προβλήματα (μεγαλύτερος όγκος δεδομένων). Η κατάκτηση του διαστήματος, οι τεχνολογικές καινοτομίες και γενικότερα η πρόοδος

των Επιστημών έκαναν πιο επιτακτική την ανάγκη αυτή. Με τον όρο ταχύτεροι υπολογιστές υπονοείται η βελτίωση των μερών που απαρτίζουν ένα υπολογιστικό σύστημα. Τόσο το λογισμικό (software) αλλά και το υλικό (hardware) είχαν κάποια φυσικά όρια. Ως γνωστών τα όρια στο υλικό και συγκεκριμένα οι βελτιώσεις στην συχνότητα λειτουργίας των επεξεργαστών των υπολογιστών (από 1980 μέχρι 2004) έφτασαν στα όρια τους. Επίσης το ότι τα θεμέλια (θεωρητικό μοντέλο λειτουργίας) της Επιστήμης της Πληροφορικής στηρίζονταν στον σειριακό υπολογισμό, αυτό ήταν από μόνο του ένα φυσικό εμπόδιο στην επίτευξη του στόχου για καλύτερους υπολογιστές. Οι απαρχές του παραλληλισμού στην Επιστήμη της Πληροφορικής χρονολογούνται από στο 1954 και μετέπειτα. Η ευρεία αποδοχή του παραλληλισμού ως μέσο για την εξέλιξη του κλάδου έγινε όταν τα μέσα (υλικό) που χρησιμοποιούνται για την κατασκευή των υπολογιστικών συστημάτων άρχισαν να αγγίζουν τα φυσικά τους όρια. Τέτοια φυσικά όρια αποτελούν η πυκνότητα των ημιαγωγών, τα φυσικά όρια της μικρογράφησης (miniaturization) και άλλα [3]. Η εμπειρία από την χρήση των σειριακών συστημάτων σε συνδυασμό με την ανάπτυξη παράλληλων συστημάτων για ακαδημαϊκούς σκοπούς και κέντρα ερευνών, οδήγησαν στην εξέλιξη του κλάδου. Στην σημερινή εποχή (2012) η πλειοψηφία των επεξεργαστών που κυκλοφορούν στην αγορά είναι πολυπύρηνοι (mutlicores). Δηλαδή πλέον σε κάθε επεξεργαστή υπάρχουν περισσότεροι από ένας πυρήνες οι οποίοι μπορούν να εκτελούν ταυτοχρόνως/παράλληλα εντολές. Από πλευράς υλικού οι αλλαγές ήταν καθοριστικές στην διαμόρφωση και του λογισμικού. Αντί την απλοποίηση του λογισμικού, οι αλλαγές αύξησαν κάπως τον βαθμό πολυπλοκότητας του λογισμικού. Χαρακτηριστικά για την υλοποίηση ενός παράλληλου προγράμματος το οποίο αξιοποιεί πλήρως τις δυνατότητες ενός πολυπύρηνου (mutlicore) επεξεργαστή απαιτεί εξειδικευμένες γνώσεις προγραμματισμού (επιπέδου Master και άνω). Εκατοντάδες ερευνητικά κέντρα ανά το παγκόσμιο χρησιμοποιούν παράλληλους υπέρ-υπολογιστές (parallel supercomputers) για την κάλυψη των αναγκών τους. Πλέον κυκλοφορούν ευρέως κινητά υπολογιστικά συστήματα (συσκευές android, ipad, κ.α.) τα οποία εμπεριέχουν πολυπύρηνους (mutlicores) επεξεργαστές. Πλέον το μέλλον έχει καθιερωθεί και είναι παράλληλο.

2.2 Παράλληλοι Αλγόριθμοι και Μετρικές

Πριν προχωρήσουμε σε περαιτέρω εμβάθυνση στους παράλληλους αλγόριθμους πρέπει να ορίσουμε τις ορολογίες ‘αλγόριθμος’ και ‘παράλληλος αλγόριθμος’.

Ο όρος ‘αλγόριθμος’ αναφέρεται σε μια ακολουθία πεπερασμένου συνόλου εντολών, αυστηρά καθορισμένων (ακολουθούν συντακτικούς κανόνες για σειριακά συστήματα) των οποίων η εκτέλεση (σε πεπερασμένο χρόνο) σε σειριακό υπολογιστικό σύστημα οδηγεί σε επιτυχή επίλυση ενός υπολογιστικού προβλήματος.

Ο όρος ‘παράλληλος αλγόριθμος’ αναφέρεται σε μια ακολουθία πεπερασμένου συνόλου εντολών, οι οποίες ακολουθούν συντακτικούς κανόνες ειδικά για παράλληλα συστήματα, των οποίων η εκτέλεση σε παράλληλο υπολογιστικό σύστημα (σε πεπερασμένο χρόνο) οδηγεί σε επιτυχή επίλυση ενός υπολογιστικού προβλήματος.

Όπως φαίνεται ξεκάθαρα από τις προηγούμενες δύο παραγράφους, υπάρχουν διαφορές στον τρόπο σύνταξης των εντολών. Έστω ότι επιθυμούμε να επιλύσουμε το ίδιο υπολογιστικό πρόβλημα σε ένα σειριακό και ένα παράλληλο υπολογιστικό σύστημα. Το σύνολο των εντολών που επιλύει επιτυχώς το πρόβλημα στο σειριακό σύστημα δεν ικανοποιεί τους συντακτικούς και λοιπούς κανόνες ούτως ώστε να εφαρμοστεί (τρέξει) στο παράλληλο υπολογιστικό σύστημα με επιτυχία (με προσαρμογή του κώδικα αυτό είναι δυνατόν, θα το δούμε στη συνέχεια). Πριν προχωρήσουμε στην παρουσίαση συγκεκριμένων απλών παραδειγμάτων κώδικα/ψευδοκώδικα θα ορίσουμε βασικές έννοιες τις οποίες θα χρησιμοποιήσουμε ευρέως στην παρούσα Ατομική Διπλωματική Εργασία.

Για να αποφανθούμε εάν ένας παράλληλος αλγόριθμος επιλύει με επιτυχία ένα συγκεκριμένο υπολογιστικό πρόβλημα πρέπει κανείς να συγκρίνει αυτό τον παράλληλο αλγόριθμο με τον καλύτερο σειριακό αλγόριθμο που επιλύει το ίδιο πρόβλημα. Εάν ο παράλληλος αλγόριθμος πετυχαίνει καλύτερες μετρήσεις (χρονικές, απόδοσης, κ.α.) τότε αυτομάτως λέμε ότι ο παράλληλος αλγόριθμος είναι καλύτερος από τον σειριακό αλγόριθμο. Όπως φαίνεται είναι απαραίτητη η χρήση κάποιων μετρικών ούτως ώστε να είναι αυτό δυνατόν.

Οι ακόλουθες μετρικές χρησιμοποιούνται για την ανάλυση της πολυπλοκότητας των αλγορίθμων [4]:

Πλήθος Επεξεργαστών: $\mathbf{P}(\boldsymbol{n}) = \boldsymbol{\rho}$

Είναι ο συνολικός αριθμός των επεξεργαστών που χρειάζονται για την επίλυση ενός προβλήματος μεγέθους n.

Παράλληλος Χρόνος Εκτέλεσης: $\mathbf{T}_{\boldsymbol{\rho}}(\boldsymbol{n})$

Είναι ο συνολικός χρόνος που απαιτείται από ρ επεξεργαστές για να επιλύσουν ένα πρόβλημα μεγέθους n.

Κόστος: $\mathbf{C}_{\boldsymbol{\rho}}(\boldsymbol{n}) = \mathbf{T}_{\boldsymbol{\rho}}(\boldsymbol{n}) \times \mathbf{P}(\boldsymbol{n})$

Αριθμητικά υπολογίζεται ως το γινόμενο του παράλληλου χρόνου εκτέλεσης επί το πλήθος των επεξεργαστών που χρησιμοποιήθηκαν.

Επίσης οι πιο κάτω μετρικές χρησιμοποιούνται για την μέτρηση της επίδοσης των αλγορίθμων:

Επιτάχυνση: $\boldsymbol{S}_{\boldsymbol{\rho}}(\boldsymbol{n}) = \frac{\boldsymbol{\tau}_{\boldsymbol{\rho}}(\boldsymbol{n})}{\mathbf{T}_{\boldsymbol{\rho}}(\boldsymbol{n})}$

όπου $\tau_{\rho}(n)$ είναι ο χρόνος εκτέλεσης του καλύτερου σειριακού αλγόριθμου

Αριθμητικά υπολογίζεται ως ο χρόνος εκτέλεσης του καλύτερου σειριακού αλγόριθμου διά τον χρόνο εκτέλεσης του παράλληλου αλγόριθμου που επιλύει το ίδιο πρόβλημα μεγέθους n.

Βάση του νόμου του Amdahl η μέγιστη δυνατή επιτάχυνση που μπορεί να πετύχει ένας παράλληλος αλγόριθμος (που χρησιμοποιεί ρ επεξεργαστές) είναι η εξής:

$$\boldsymbol{S}_{\boldsymbol{\rho}}(\boldsymbol{n}) = \frac{\mathbf{1}}{\boldsymbol{k} + \frac{(\mathbf{1}-\boldsymbol{k})}{\boldsymbol{\rho}}}$$

όπου k το κλάσμα του υπολογισμού που είναι υποχρεωτικά σειριακό και δύναται να πάρει τιμές από μηδέν μέχρι ένα $(0 \leq k \leq 1)$.

Όταν υπάρχει πλήρης παραλληλοποίηση του σειριακού αλγορίθμου τότε έχουμε $S = \Theta(\rho)$ και λέμε ότι ο αλγόριθμος επιτυγχάνει γραμμική επιτάχυνση.

Απόδοση Κόστους (cost efficiency): $\boldsymbol{e_\rho(n) = \frac{\tau_\rho(n)}{C_\rho(n)} = \frac{\tau_\rho(n)}{T_\rho(n) \times P(n)} = \frac{S_\rho(n)}{\rho} \leq 1}$

όπου $\tau_\rho(n)$ είναι ο χρόνος εκτέλεσης του καλύτερου σειριακού αλγόριθμου

Αριθμητικά υπολογίζεται ως ο χρόνος εκτέλεσης του καλύτερου σειριακού αλγόριθμου διά το κόστος του παράλληλου αλγόριθμου που επιλύει το ίδιο πρόβλημα μεγέθους n. Η απόδοση κόστους μας δείχνει το πόσο καλά αξιοποιούμε τους επεξεργαστές στο παράλληλο υπολογιστικό σύστημα κατά την εφαρμογή/λειτουργία του παράλληλου αλγόριθμου.

Στο σημείο αυτό θα μελετήσουμε ένα σχετικά απλό παράδειγμα για να δούμε κάποια εφαρμογή των εννοιών που προαναφέραμε (είναι εξαιρετικά απλό και βοηθά αρκετά στην κατανόηση των βασικών εννοιών των Παράλληλων Αλγορίθμων).

Πρόβλημα: Έστω ένα σύνολο εργασιών Ε το οποίο αποτελείται από n υπό-εργασίες $E = \{\varepsilon_1, \varepsilon_2, \dots, \varepsilon_n\}$ οι οποίες στο σύνολο τους πρέπει να ολοκληρωθούν στο μικρότερο χρονικό διάστημα που είναι δυνατόν (δηλαδή να μειωθεί ο χρόνος ολοκλήρωσης του συνόλου εργασιών Ε στο ελάχιστο). Υποθέτουμε ότι για την ολοκλήρωση της οποιασδήποτε εργασίας $\varepsilon_i \in E$, όπου $1 \leq i \leq n$, απαιτείται μία (1) μονάδα χρόνου.

Λύση:

Πιο κάτω παρουσιάζονται οι ψευδοκώδικες που επιλύουν το πρόβλημα τόσο σε σειριακό αλλά και παράλληλο σύστημα:

Σειριακός ψευδοκώδικας:

```
i := 1
Εφόσον το i <= n κάνε:
     Ολοκλήρωσε την υπό-εργασία εi
     i := i + 1
νοσόφΕ
```

$P(n) = \rho = 1$

$$T_1(n) = \Theta(n)$$

$$C_1(n) = T_1(n) \times P(n) = \Theta(n) \times 1 = \Theta(n)$$

Παράλληλος ψευδοκώδικας:

```
Εκτέλεσε παράλληλα για ρi := 1…n
     Ολοκλήρωσε την υπό-εργασία εi
εσελέτκΕ
```

$$P(n) = \rho = n$$

$$T_1(n) = \Theta(1)$$

$$C_1(n) = T_1(n) \times P(n) = \Theta(1) \times n = \Theta(n)$$

Παρατηρούμε ότι ο σειριακός αλγόριθμος χρησιμοποιεί ένα (1) επεξεργαστή προκειμένου να ολοκληρώσει τον φόρτο εργασίας του ενώ ο παράλληλος αλγόριθμος χρησιμοποιεί n επεξεργαστές. Μιλώντας πιο συγκεκριμένα για τον παράλληλο αλγόριθμο θα πούμε ότι ο κάθε ένας από τους n επεξεργαστές ολοκληρώνει ξεχωριστή υπό-εργασία. Δηλαδή η σχέση του συνόλου επεξεργαστών που χρησιμοποιούνται με το σύνολο των υπό-εργασιών είναι 1:1. Επίσης να τονίσουμε το γεγονός ότι οι εργασίες είναι ανεξάρτητες, δηλαδή η εκτέλεση κάποιας εργασίας δεν εξαρτάται από το αποτέλεσμα κάποιας άλλης.

Πλέον σε αυτό το σημείο είμαστε έτοιμοι να κάνουμε αναφορά για την <u>σύγκριση αποδοτικότητας μεταξύ δύο παράλληλων αλγορίθμων</u>. Η έννοια του κόστους χρησιμοποιείται σαν βασικό μέτρο σύγκρισης μεταξύ δύο παράλληλων αλγορίθμων. Έστω ότι οι δύο διαφορετικοί παράλληλοι αλγόριθμοι (Α και Β), επιλύουν το ίδιο υπολογιστικό πρόβλημα μεγέθους (n) με κόστη $C_A(n)$ και $C_B(n)$ σε χρόνους $T_A(n)$ και $T_B(n)$. Τότε εφαρμόζουμε τα εξής:

- Θα λέμε ότι ο παράλληλος αλγόριθμος Α είναι αποδοτικότερος από τον Β αν $C_A(n) = O\big(C_B(n)\big)$ ανεξαρτήτως του χρόνου ολοκλήρωσης.
- Εάν όμως $C_A(n) = \Theta\big(C_B(n)\big)$ τότε ο παράλληλος αλγόριθμος Α είναι πιο αποδοτικός από τον Β εάν $T_A(n) = O\big(T_B(n)\big)$.

Επίσης χρησιμοποιούμε την ορολογία κόστους-βέλτιστος αλγόριθμος για να χαρακτηρίσουμε ένα παράλληλο αλγόριθμο, που επιλύει ένα πρόβλημα μεγέθους n, όταν $C_\rho(n) = O\left(\tau_\rho(n)\right)$, όπου $\tau_\rho(n)$ ο χρόνος της καλύτερης σειριακής εκτέλεσης. Επίσης όταν το $T_\rho(n)$ δεν δίνατε να ελαχιστοποιηθεί περισσότερο τότε ο παράλληλος αλγόριθμος μας μπορεί να χαρακτηριστεί και χρόνου-βέλτιστος. Ο παράλληλος αλγόριθμος που επιλύει το προαναφερθέν πρόβλημα υλοποίησης υπό-εργασιών είναι κόστους-βέλτιστος και χρόνου-βέλτιστος.

2.3 Μοντέλα - Αρχιτεκτονικές Υπολογισμού

Πριν αναφερθούμε συγκεκριμένα για το μοντέλο παράλληλου υπολογισμού που μοντελοποιεί η πλατφόρμα XMT είναι ορθό να μιλήσουμε για τις κατηγορίες των μοντέλων-αρχιτεκτονικών υπολογισμού γενικά ούτως ώστε να κατανοήσουμε καλύτερα τα θετικά και τα αρνητικά του συγκεκριμένου μοντέλου που θα μελετήσουμε.

Το 1966 ο Michael J. Flynn πρότεινε την κατηγοριοποίηση των αρχιτεκτονικών υπολογισμού [5] χρησιμοποιώντας ως μέτρα κατηγοριοποίησης τις παραμέτρους ‘ρεύμα εντολών’ και ‘ρεύμα δεδομένων’. Ο όρος ‘ρεύμα εντολών’ (instruction stream) αναφέρεται στο σύνολο των εντολών που απαρτίζουν το αλγόριθμο μας. Ο όρος ‘ρεύμα δεδομένων’ (data stream) αναφέρεται στο σύνολο των δεδομένων που επεξεργάζεται ο αλγόριθμος μας, δηλαδή τα δεδομένα εισόδου του αλγορίθμου μας. Ο λόγος για τον οποίο χρησιμοποίηση των δύο όρων ως μέτρα κατηγοριοποίησης οφείλεται στο ότι όλα τα υπολογιστικά συστήματα (σειριακά ή παράλληλα) εκτελούν σύνολα εντολών σε σύνολα δεδομένων. Ο Πίνακας 2.1 παρουσιάζει τις σχέσεις μεταξύ των δύο μέτρων κατηγοριοποίησης.

	Single instruction	**Multiple instruction**
Single data	SISD	MISD
Multiple data	SIMD	MIMD

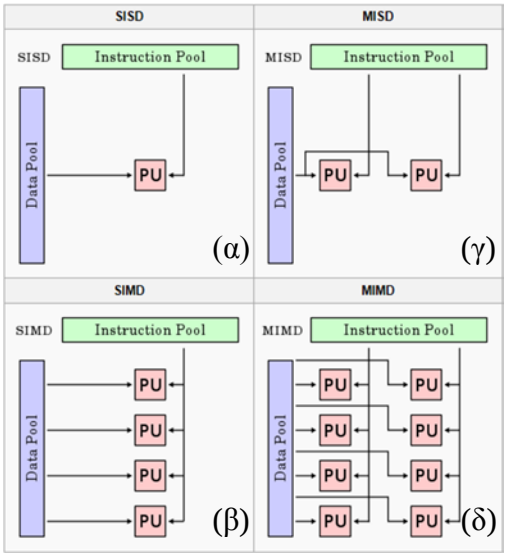
Πίνακας 2.1 – Κατηγοριοποίηση του Flynn [2]

Οι τέσσερις κατηγορίες που πρότεινε ήταν οι εξής:

- Single Instructions set, Single Data stream (SISD): Σε αυτή την κατηγορία ανήκουν οι σειριακοί υπολογιστές. Δηλαδή μοντέλο αυτό υπάρχει μόνο ένας

επεξεργαστής ο οποίος φορτώνει μόνο ένα ‘ρεύμα εντολών’ κάθε φορά από την μνήμη και ακολούθως εκτελεί βάσει του συγκεκριμένου ρεύματος πράξεις σε μόνο ένα ‘ρεύμα δεδομένων’. Για διαγραμματική απεικόνιση του μοντέλου SISD βλέπε Σχήμα 2.1(α).

- Single Instructions set, Multiple Data stream (SIMD): στην κατηγοριοποίηση αυτή υπάρχουν περισσότεροι από ένας επεξεργαστές οι οποίοι έχουν ένα κοινό ‘ρεύμα εντολών’ και ο κάθε ένας επεξεργάζεται ένα ξεχωριστό ‘ρεύμα δεδομένων’. Όπως φαίνεται και στο Σχήμα 2.1(β) υπάρχει μια κοινή πηγή του ‘ρεύματος εντολών’ αλλά χρειάζεται προσοχή αφού τοα ‘ρεύματα δεδομένων’ λαμβάνουν χώρα πλέον με την χρήση κοινόχρηστης μνήμης ή ενός δικτύου αλληλοσύνδεσης. Εξίσου σημαντικό είναι και το γεγονός ότι οι επεξεργαστές λειτουργούν συγχρονισμένα, δηλαδή σε κάθε βήμα όλοι οι επεξεργαστές που εκτελούν το ίδιο ‘ρεύμα εντολών’ βρίσκονται στην ίδια εντολή (αλλά πάνω σε διαφορετικό ‘ρεύμα δεδομένων’).
- Multiple Instructions sets, Single Data stream (MISD): η κατηγορία αυτή εκτελεί πολλαπλά ‘ρεύματα εντολών’ πάνω στον ίδιο ‘ρεύμα δεδομένων’. Συγκεκριμένα αρχιτεκτονικές αυτής της κατηγορίας αποτελούνται από περισσότερους από ένα επεξεργαστές οι οποίοι επενεργούν πάνω στο ίδιο ‘ρεύμα δεδομένων’. Κάθε επεξεργαστής έχει το δικό του ξεχωριστό ‘ρεύμα εντολών’. Διαγραμματική αναπαράσταση του μοντέλου φαίνετε στο Σχήμα 2.1(γ). Αν και δεν υπάρχουν υλοποιημένα αρκετά παραδείγματα αυτής της κατηγορίας, υπάρχει μια υποσχόμενη υλοποίηση του μοντέλου MISD η οποία χρησιμοποιείται στους υπολογιστές των συστημάτων ελέγχου πτήσεων του διαστημικού λεωφορείου (Space Shuttle flight control computers) [5] η οποία αξίζει προσοχής. Ο λόγος για τον οποίο δεν υπάρχουν αρκετές υλοποιήσεις είναι εξαιτίας του ότι οι κατηγορίες-αρχιτεκτονικές MIMD



Σχήμα 2.1 – Διαγραμματική απεικόνιση μοντέλων για τις κατηγοριοποιήσεις του Flynn [2]

και SIMD είναι πιο προσιτές για την εφαρμογή κοινών τεχνικών παράλληλης επεξεργασία δεδομένων.

- Multiple Instructions sets, Multiple Data stream (MIMD): στην συγκεκριμένη κατηγορία συναντούμε το πιο γενικό και δυνατό μοντέλο παράλληλου υπολογισμού. Αυτό συμβαίνει γιατί επιτρέπει σε πολλαπλούς επεξεργαστές να εκτελούν πολλαπλά 'ρεύματα εντολών' σε πολλαπλά 'ρεύματα δεδομένων'. Ας το κάνουμε πιο ξεκάθαρο. Έστω ότι ένα παράλληλο υπολογιστικό σύστημα κατηγορίας MIMD εμπεριέχει n επεξεργαστές. Αυτό υποδηλώνει ότι το συγκεκριμένο σύστημα είναι σε θέση να εκτελεί n διαφορετικά προγράμματα πάνω σε n διαφορετικά δεδομένα με αποτέλεσμα να επιλύουν έως και n διαφορετικά προβλήματα. Το αξιοσημείωτο είναι ότι τα n διαφορετικά προβλήματα μπορεί να αποτελούν υπό-προβλήματα ενός ακόμη μεγαλύτερου προβλήματος. Επίσης το συγκεκριμένο μοντέλο επιτρέπει στους επεξεργαστές να εργάζονται ασυγχρόνιστα. Παρ' όλα τα θετικά που μας παρέχει υπάρχουν και κάποια αρνητικά στοιχεία, όπως η αυξημένη πολυπλοκότητα στον σχεδιασμό και ανάλυση αλγορίθμων στο μοντέλο αυτό. Διαγραμματική απεικόνιση του μοντέλου αυτού φαίνεται στο Σχήμα 2.1(δ).

2.4 Μοντέλα Παράλληλου Υπολογισμού

Τα μοντέλα PRAM και F-PRAM, ανήκουν στην κατηγορία SIMD. Το μοντέλο A-PRAM ανήκει στην κατηγορία MIMD. Όπως προαναφέραμε (βλέπε υποκεφάλαιο 2.3) η κατηγορία SIMD χρησιμοποιεί πολλαπλά 'ρεύματα δεδομένων' (Multiple Data streams) με την χρήση Κοινόχρηστης Μνήμης είτε Δικτύου Αλληλοσύνδεσης. Τα μοντέλα αυτά χρησιμοποιούν Κοινόχρηστη Μνήμη. Προτού προχωρήσουμε στην ανάλυση των μοντέλων ας κατανοήσουμε το βασικό μοντέλο PRAM το οποίο θα μας υποβοηθήσει τόσο στην εξήγηση του μοντέλου F-PRAM. Στην συνέχεια θα αναλύσουμε το μοντέλο A-PRAM.

2.4.1 Μοντέλο Παράλληλου Υπολογισμού PRAM

Η ακριβής ορολογία του PRAM είναι Parallel Random Access Machine (Παράλληλη Μηχανή Τυχαίας Προσπέλασης) [6]. Οι αρχές που διέπουν το συγκεκριμένο μοντέλο – μηχανή είναι οι εξής:

- Η παράλληλη μηχανή αποτελείται από ρ όμοιους και συγχρονισμένους επεξεργαστές.
- Κάθε επεξεργαστής διαθέτει την δική του τοπική μνήμη στην οποία ο χρόνος πρόσβασης είναι $\Theta(1)$.
- Η κοινόχρηστη μνήμη χρησιμοποιείται ως ‘γέφυρα’ επικοινωνίας μεταξύ των επεξεργαστών. Σε αυτή βρίσκονται τα δεδομένα εισόδου, αποθηκεύονται ενδιάμεσα αποτελέσματα αλλά επίσης και οι πληροφορίες εξόδου. Η πρόσβαση στην κοινόχρηστη μνήμη γίνεται διαμέσου προκαθορισμένων κοινόχρηστων δομών (μεταβλητές, κ.α.) με χρόνο πρόσβασης $\Theta(1)$.

Το μοντέλο PRAM μπορεί να ομαδοποιηθεί με βάση τους περιορισμούς που εμφανίζονται κατά την ταυτόχρονη πρόσβαση στην κοινόχρηστη μνήμη. Με τον όρο ταυτόχρονη πρόσβαση εννοούμε τις έννοιες της ταυτόχρονης ανάγνωσης και γραφής δεδομένων / πληροφοριών από και προς την κοινόχρηστη μνήμη. Οι τέσσερις τύποι που δημιουργούνται είναι:

- Exclusive Read, Exclusive Write (EREW): Σε αυτή την ομάδα ανήκουν τα μοντέλα όπου μόνο ένας επεξεργαστής μπορεί να διαβάσει ή να γράψει σε μια συγκεκριμένη θέση / κυψελίδα μνήμης.
- Concurrent Read, Exclusive Write (CREW): η ομάδα αυτή χαρακτηρίζεται από την ιδιότητα ότι μπορούν όλοι οι επεξεργαστές να διαβάσουν από μία συγκεκριμένη θέση / κυψελίδα μνήμης, όμως μόνο ένας επεξεργαστής μπορεί να γράψει στην συγκεκριμένη θέση / κυψελίδα.
- Exclusive Read, Concurrent Write (ERCW): Σε αυτή την ομάδα μπορούν όλοι οι επεξεργαστές να γράψουν σε μια συγκεκριμένη θέση / κυψελίδα μνήμης, όμως μόνο ένας επεξεργαστής μπορεί να διαβάσει από μια συγκεκριμένη θέση / κυψελίδα μνήμης. Αυτή η ομάδα δεν έχει υλοποιηθεί και απλώς αναφέρεται εξαιτίας της ομαδοποίησης που προκύπτει βάση των χαρακτηριστικών της ανάγνωσης και γραφής στην μνήμη.

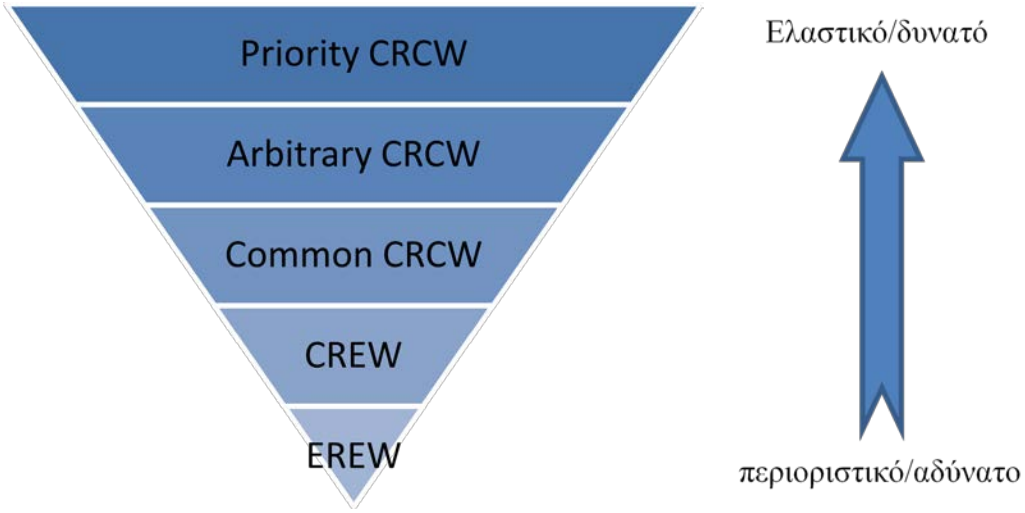
- Concurrent Read, Concurrent Write (CRCW): Αυτή η ομάδα δεν θέτει περιορισμούς στην ταυτόχρονη πρόσβαση στην μνήμη (αυτό ισχύει για την ανάγνωση και την γραφή).

Από τους προαναφερθέντες τύπους / ομάδες του PRAM προκύπτει το ζήτημα της ταυτοχρονίας κατά την πρόσβαση στην κοινόχρηστη μνήμη. Κατά την ανάγνωση δεδομένων από συγκεκριμένη θέση / κυψελίδα μνήμης από πολλούς επεξεργαστές δεν προκύπτει καθόλου ζήτημα Το πρόβλημα προκύπτει κατά την ταυτόχρονη γραφή δεδομένων ή πληροφοριών από πολλούς επεξεργαστές σε μια συγκεκριμένη θέση / κυψελίδα μνήμης. Για την καλύτερη κατανόηση του προβλήματος υποθέστε το σενάριο που πολλαπλοί επεξεργαστές προσπαθούν να γράψουν σε μια συγκεκριμένη θέση μνήμης το αποτέλεσμα που έχουν υπολογίσει. Έστω ότι όλοι οι επεξεργαστές έχουν υπολογίσει διαφορετικό αποτέλεσμα. Όπως φαίνεται δεν είναι εύκολο μετά που ολοκληρώνουν το γράψιμο στην κοινόχρηστη μνήμη οι επεξεργαστές, να απαντήσει κανείς ποια τιμή βρίσκεται αποθηκευμένη στην συγκεκριμένη θέση μνήμης. Για τον λόγο ο τύπος CRCW του PRAM διακρίνεται σε τρείς εκδόσεις ως εξής:

i. Common CRCW PRAM: στην έκδοση αυτή για να επιτραπεί το γράψιμο σε μια θέση μνήμης, πρέπει όλοι οι επεξεργαστές (που επιθυμούν να γράψουν στην συγκεκριμένη θέση μνήμης) να γράψουν την ίδια τιμή.
ii. Arbitrary CRCW PRAM: σε αυτή την έκδοση θεωρείτε ότι ένας τυχαίος επεξεργαστής έχει επιτύχει να αποθηκεύσει το αποτέλεσμα του στην συγκεκριμένη θέση μνήμης.
iii. Priority CRCW PRAM: ενώ στην έκδοση αυτή ακολουθείται προτεραιότητα για την αποθήκευση του αποτελέσματος στην συγκεκριμένη θέση μνήμης, δηλαδή μόνο ο επεξεργαστής με την μεγαλύτερη προτεραιότητα ολοκληρώνει με επιτυχία την αποθήκευση του αποτελέσματος.

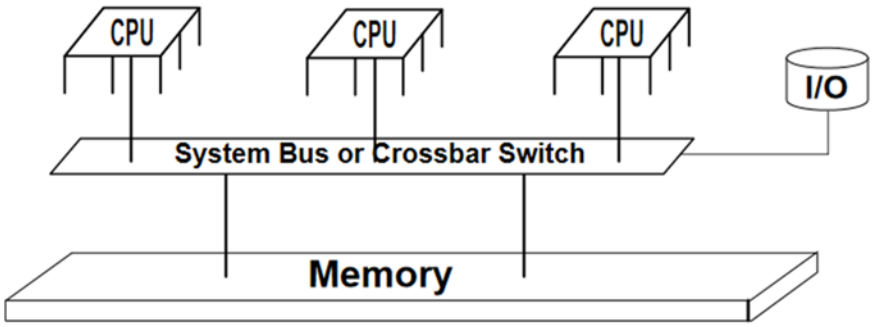
Παρατηρούμε ότι οι διαφορετικές κατηγοριοποιήσεις του PRAM, δηλαδή τα υπό-μοντέλα του PRAM μπορούν με βάση την δυναμικότητα τους να ιεραρχηθούν. Στο Σχήμα 2.2 (δες επόμενη σελίδα) παρουσιάζεται αυτή η ιεράρχηση. Είναι δυνατή η εκτέλεση αλγόριθμου που είναι σχεδιασμένος για ένα αδύνατο μοντέλο σε ένα πιο δυνατό μοντέλο με την ίδια πολυπλοκότητα κόστους και χρόνου. Επίσης μπορεί να συμβεί και το αντίθετο, από ένα δυνατό μοντέλο σε ένα αδύνατο, αλλά αυτό γίνεται με

την χρήση ασυμπτοτικά μεγαλύτερου αριθμού επεξεργαστών ή με ασυμπτοτικά μεγαλύτερο χρόνο εκτέλεσης.



Σχήμα 2.2 – Ιεραρχία δυναμικότητας τύπων PRAM

Ολοκληρώνοντας την αναφορά στο μοντέλο PRAM αναφέρουμε ότι η αρχιτεκτονική που μοντελοποιεί το PRAM είναι η SMP (Symmetric Multiprocessing). Αυτή η αρχιτεκτονική δομή παρουσιάζεται στο Σχήμα 2.3 και τα κύρια συστατικά της στοιχεία είναι οι επεξεργαστές (δύο ή περισσότεροι), ο κοινός δίαυλος επικοινωνίας, η κοινόχρηστη μνήμη και η κοινόχρηστη μονάδα είσοδο/εξόδου (I/O) [7]. Λαμβάνοντας υπόψη το Σχήμα 2.1(β) μπορείτε να παρατηρήσετε την διατήρηση της δομής του μοντέλου SIMD (για περισσότερες πληροφορίες δες υποκεφάλαιο 2.3). Χαρακτηριστικά να αναφέρουμε επίσης ότι η αρχιτεκτονική δομή SMP είναι ιδιαίτερα δημοφιλής στην σημερινή εποχή όπου οι πολυπύρηνοι (multicores) επεξεργαστές βρίσκονται σε κυρίαρχη θέση στην αγορά.



Σχήμα 2.3 – Αφαιρετική αναπαράσταση της αρχιτεκτονικής δομής του SMP

2.4.2 Μοντέλο Παράλληλου Υπολογισμού F-PRAM

Στην περίπτωση του μοντέλου F-PRAM τα πράγματα είναι πολύ πιο απλά σε σχέση με το A-PRAM. Το μοντέλο F-PRAM είναι το ίδιο ακριβώς με το PRAM με την μόνη διαφορά ότι οι επεξεργαστές υπόκεινται σε σφάλματα (failures). Με τον όρο σφάλμα, εννοούμε την ξαφνική και τυχαία κατάρρευση ενός επεξεργαστή. Δεδομένου ενός αλγορίθμου που επιλύει ένα πρόβλημα στο μοντέλο F-PRAM και κάνει χρήση (ρ) επεξεργαστών, ο μέγιστος αριθμός των διαφορετικών σφαλμάτων που μπορούν να συμβούν είναι $(\rho - 1)$. Αυτός ο περιορισμός υπάρχει για να διασφαλίζει την ολοκλήρωση της λύσης του προβλήματος. Το πλήθος των σφαλμάτων που προκύπτουν συμβολίζεται ως (f), όπου $0 \leq f < (\rho - 1)$.

Στο μοντέλο F-PRAM χρησιμοποιείτε η ορολογία εύρωστος παράλληλος αλγόριθμος, για να χαρακτηρίσει ένα παράλληλο αλγόριθμο ο οποίος επιλύει αποδοτικά το πρόβλημα και συνάμα ανέχεται σφάλματα επεξεργαστών. Η επίτευξη της ευρωστίας κατά τον σχεδιασμό ενός παράλληλου αλγορίθμου στο F-PRAM είναι δύσκολη γιατί για να αυξηθεί η απόδοση του αλγορίθμου πρέπει να μειωθεί ο χρόνος εκτέλεσης ή το πλήθος των επεξεργαστών. Επίσης πρέπει να υπάρχει σημαντικός βαθμός ανοχής σφαλμάτων πράγματα το οποίο προϋποθέτει αύξηση του χρόνου εκτέλεσης ή αύξηση του πλήθους των επεξεργαστών. Όπως έχετε προσέξει οι δύο τελευταίες προτάσεις δείχνουν προς δύο αντίθετες κατευθύνσεις συνεπώς και η δυσκολία στον σχεδιασμό εύρωστων παράλληλων αλγορίθμων στο F-PRAM.

Ολοκληρώνοντας την αναφορά μας στο μοντέλο F-PRAM θα αναφέρουμε και τις διαφορές στον τρόπο υπολογισμού του παράλληλου χρόνου εκτέλεσης και του κόστους.

Παράλληλος Χρόνος Εκτέλεσης: $\mathbf{T}_{\boldsymbol{\rho}}(\boldsymbol{n}, \boldsymbol{f})$

> Είναι ο συνολικός χρόνος που απαιτείται από (ρ) επεξεργαστές για να επιλυθεί ένα πρόβλημα μεγέθους (n), με την παρουσία (f) σφαλμάτων κατάρρευσης (όπου $0 \leq f < (\rho - 1)$).

Κόστος: $\mathbf{C}_{\boldsymbol{\rho}}(\boldsymbol{n}, \boldsymbol{f}) = \sum_{\boldsymbol{\tau=1}}^{\boldsymbol{\beta}} \boldsymbol{\pi}_{\boldsymbol{\tau}}$

Υποθέτουμε ότι το πρόβλημα λύνεται σε (β) βήματα. Έστω (π_τ) το πλήθος των επεξεργαστών που δεν έχουν καταρρεύσει μέχρι το τέλος του βήματος (τ). Ισχύει ότι $\left(\pi_0 = \rho \geq \pi_1 \geq \pi_2 \geq \cdots \geq \pi_\beta\right)$, το (π_τ) φθίνει.

2.4.3 Μοντέλο Παράλληλου Υπολογισμού A-PRAM

Όπως τονίσαμε στην εισαγωγή του υποκεφαλαίου 2.4, θα παρουσιάζουμε και το μοντέλο παράλληλου υπολογισμού A-PRAM. Η πλήρης ερμηνεία του όρου A-PRAM είναι Asynchronous Parallel Random Access Machine. Υπάρχουν ομοιότητες σε μεγάλο βαθμό με το μοντέλο PRAM αλλά υπάρχουν οι εξής διαφορές:

- Πλέον οι επεξεργαστές **δεν** είναι συγχρονισμένοι.
- Επίσης είναι δυνατόν οι επεξεργαστές να έχουν πρόσβαση (ανάγνωση ή εγγραφή) σε οποιαδήποτε θέση/κυψελίδα στην κοινόχρηστη μνήμη όμως κάθε φορά η πρόσβαση αυτή μπορεί να χρειαστεί διαφορετικό χρόνο ολοκλήρωσης.
- Σε αυτό το μοντέλο εισάγεται και η έννοια της ατομικότητας, η οποία επηρεάζει την έννοια της ταυτόχρονης πρόσβασης στην μνήμη. Συγκεκριμένα στο A-PRAM δεν υπάρχουν ταυτόχρονες προσβάσεις σε συγκεκριμένη θέση μνήμης. Η έννοια της ατομικότητας επιβάλλει την μερική διάταξη των προσβάσεων από πολλαπλούς επεξεργαστές σε συγκεκριμένη θέση μνήμης ώστε να διασφαλιστούν οι αρχές τις ατομικότητας:
    - Η τιμή μιας ανάγνωσης της κυψελίδας να είναι η τιμή της τελευταίας ολοκληρωμένης εγγραφής ή της τιμής μιας γραφής που συμβαίνει το ίδιο χρονικό διάστημα (και δεν έχει ολοκληρωθεί)
    - Η τιμή της επόμενης ανάγνωσης της κυψελίδας μνήμης να είναι η ίδια με την προηγούμενη ανάγνωση είτε η τιμή μιας γραφής που λαμβάνει χώρα μετά την προηγούμενη ανάγνωση.

Μπορεί κανείς να παρατηρήσει ότι οι διαφορές είναι τέτοιες ώστε εύλογα διερωτάται κατά πόσον επηρεάζεται το παράλληλος χρόνος ολοκλήρωσης ή το κόστος. Καταρχάς σε αυτό το μοντέλο δεν παίζει σημαντικό ρόλο ο χρόνος αφού οι επεξεργαστές λειτουργούν εντελώς ασυγχρόνιστα. Υπάρχει όμως διαφορά στον τρόπο υπολογισμού του κόστους ενός παράλληλου αλγορίθμου στο A-PRAM. Το κόστος μπορεί να υπολογιστεί με την εφαρμογή του τύπου:

$$\mathbf{C'}_{\boldsymbol{\rho}}(\boldsymbol{n}) = \sum_{\boldsymbol{i=1}}^{\boldsymbol{\rho}} \boldsymbol{c_i}$$

, όπου (c_i) είναι ο συνολικός αριθμός μονάδων κόστους που χρεώνεται ο επεξεργαστής (i) μέχρι να λυθεί ένα πρόβλημα μεγέθους (n). Η ερμηνεία της μονάδας κόστους έχει ως εξής· κάθε επεξεργαστής χρεώνεται μία μονάδα κόστους για κάθε διάβασε-υπολόγισε-γράψε πράξη που ολοκληρώνει. Με πιο απλά λόγια η πράξη αυτή περιλαμβάνει την ανάγνωση από την μνήμη (read/fetch data), την εκτέλεση υπολογισμών βάση των δεδομένων αυτών και την εγγραφή σε θέση μνήμης (write/store data).

Τελειώνοντας την αναφορά στο μοντέλο A-PRAM πρέπει αναφέρουμε ότι εφόσον οι επεξεργαστές ενεργούν εντελώς ασυγχρόνιστα επαφίεται στον προγραμματιστή να αποφύγει την απροσδιοριστία αποτελεσμάτων που προκαλεί η ατομικότητα. Ακολουθεί ένα απλό παράδειγμα με στόχο να θίξει αυτό το πρόβλημα.

παράδειγμα:

```
KM[1] := 0
Processors ρ1 , ρ2 do asychronously

     ρ1:  KM[1] := KM[1] + 1

     ρ2:  KM[1] := KM[1] + 1

od
```

Εάν θέταμε το ερώτημα ποια είναι η τιμή που βρίσκεται αποθηκευμένη στην κυψελίδα KM[1], ποια θα ήταν η ορθή απάντηση· ένα ή δύο; Η απάντηση είναι ότι και οι δύο τιμές είναι πιθανές απαντήσεις. Αυτό συμβαίνει εξαιτίας του ότι οι επεξεργαστές είναι ασυγχρόνιστοι.

Υπάρχουν πιθανότητες να συμβεί κάποιο από τα επόμενα σενάρια που θα αναφέρω, υπάρχουν και άλλες περιπτώσεις απλώς δεν τις αναφέρουμε γιατί μπορούν να γίνουν αναγωγή σε κάποιο από τα σενάρια αυτά. Κατά το πρώτο σενάριο διαβάζουν και οι δύο επεξεργαστές ταυτόχρονος την τιμή μηδέν και εκτελούν την πράξη. Ακολούθως αποθηκεύουν, λαμβάνοντας υπόψη την ατομικότητα, στην μνήμη το αποτέλεσμα. Και οι δύο επεξεργαστές γράφουν την τιμή 1 στην κυψέλη KM[1].

Το δεύτερο σενάριο εμπεριέχει το γεγονός ότι ο ένας από τους δύο επεξεργαστές είναι υπερβολικά αργός και ο άλλος υπερβολικά ταχύς. Έστω ότι ο γρήγορος επεξεργαστής καταφέρνει να ολοκληρώσει την εργασία του δηλαδή να διαβάσει την τιμή της κυψελίδα ΚΜ[1] (η οποία εμπεριέχει την τιμή μηδέν), να εκτελέσει την πράξη και να αποθηκεύσει το αποτέλεσμα στη θέση ΚΜ[1] πριν ο αργός επεξεργαστής διαβάσει την τιμή της κυψελίδας ΚΜ[1]. Όταν ο αργός επεξεργαστής καταφέρνει να διαβάσει την τιμή της κυψελίδας ΚΜ[1] παραλαμβάνει την τιμή ένα. Επομένως με την ολοκλήρωση της εργασίας του αργού επεξεργαστή, στην κυψελίδα μνήμης ΚΜ[1] θα βρίσκεται αποθηκευμένη η τιμή 2.

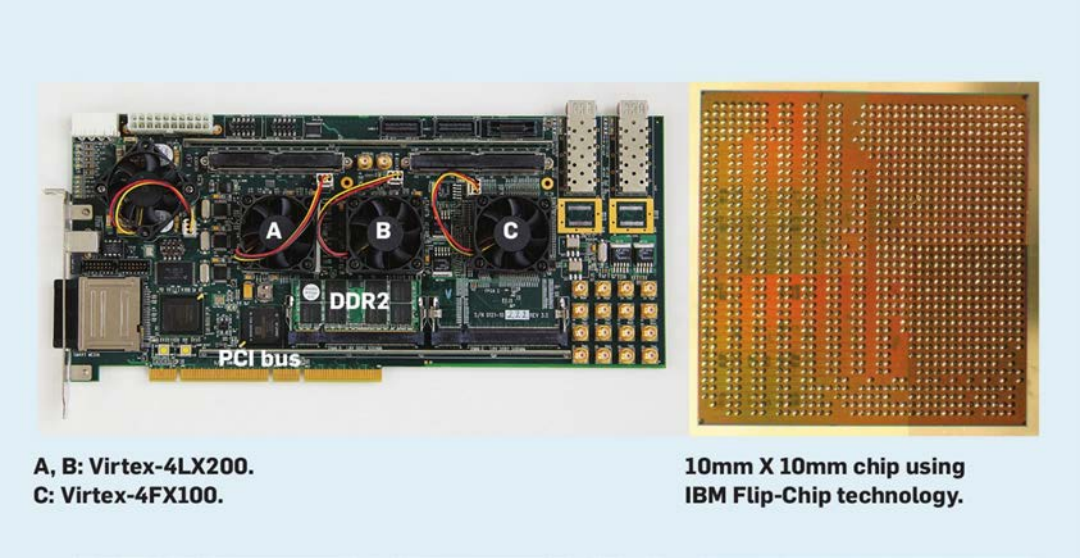
Θα επαναλάβουμε το γεγονός ότι ο προγραμματιστής έχει την ευθύνη να λάβει υπόψη του την ατομικότητα ούτος ώστε να αποφύγει την απροσδιοριστία αποτελεσμάτων, γιατί αποτελεί κομβικό σημείο στην ορθότητα των παράλληλων αλγορίθμων του μοντέλου A-PRAM.

Κεφάλαιο 3

Πλατφόρμα XMT

3.1 Ιστορία

Η σύλληψη της ιδέας του XMT (Explicit Multi-Threading) είχε γίνει το 1997. Το επόμενο έτος (1998) είχε παρουσιαστεί η ιδέα του XMT από τον καθηγητή Uzi Vishkin στο Πανεπιστήμιο του Maryland [2] [8] [9]. Τα επόμενα χρόνια ήταν καθοριστικά εξαιτίας του ότι η ομάδα υπεύθυνη για την έρευνα και ανάπτυξη του XMT, υπό την καθοδήγηση του καθηγητή Uzi Vishkin, κατάφερε το 2007 να παρουσιάσει την υλοποίηση ενός πρωτότυπου PRAM-On-Chip, 64-core, 75MHz FPGA (βλέπε Σχήμα 3.1). Μέχρι σήμερα η ομάδα σε συνεργασία με άλλα πανεπιστήμια συνεχίζει την περαιτέρω μελέτη, ανάπτυξη και βελτίωση του XMT.



**Σχήμα 3.1 – Αριστερά: FPGA board με τρία FPGA chips (A,B και C).
Δεξιά: chip με τεχνολογία IBM Flip-Chip.**

Φυσικά το πεδίο των παράλληλων αλγορίθμων προϋπήρχε και σαφώς ήταν ήδη πλούσιο αλγοριθμικά. Απλώς η εξέλιξη της τεχνολογίας έκανε δυνατή την υλοποίηση ενός επεξεργαστή που έχει ως βάση το μοντέλο υπολογισμού PRAM.

3.2 Μοντέλο Παράλληλου Υπολογισμού XMT

Το μοντέλο παράλληλου υπολογισμού που υλοποιεί η πλατφόρμα XMT είναι το arbitrary CRCW Single Program Multiple Data - SPMD (εν μέρει όμοιο με SIMD) με εφαρμογή της μεθοδολογίας Work-Depth (WD), το συζητάμε πιο κάτω, κατά την δημιουργία του αλγορίθμου-κώδικα. Ο λόγος για τον οποίο χρησιμοποιείται η μεθοδολογία Work-Depth είναι εξαιτίας του ασυγχρονισμού που εμφανίζεται στην πλατφόρμα, δηλαδή οι επεξεργαστές (TCUs) δεν έχουν συγχρονισμένα βήματα και είναι αναγκαία η μερική μετατροπή του κώδικα για να ανταποκρίνεται στην ιδιαιτερότητα αυτή της πλατφόρμας XMT.

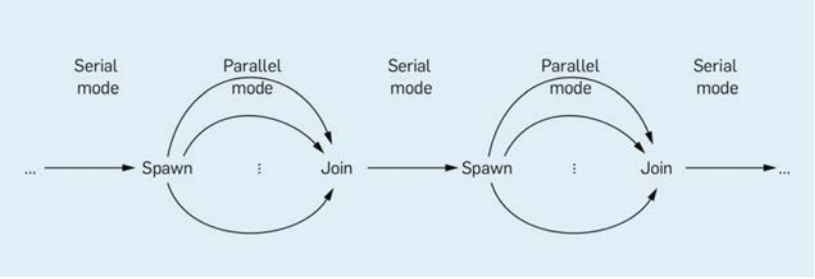
Ιστορικά κατά τις προσπάθειες ανάπτυξης ενός PRAM επεξεργαστή, οι ομάδες που εργάζονταν στο αντικείμενο κατέληγαν πάντα στο συμπέρασμα ότι δεν ήταν δυνατή η δημιουργία ενός επεξεργαστή που να ανταποκρίνεται ακριβώς στο μοντέλο PRAM, εξαιτίας του μεγάλου και ασύμφορου κόστους για τον αυστηρό συγχρονισμό των επεξεργαστών. Αυτό το οποίο έθεσε εν μέρη τα θεμέλια για την ανάπτυξη του XMT processor ήταν το γεγονός της μερικής χαλάρωσης στον κανόνα που ορίζει τον αυστηρό συγχρονισμό μεταξύ των βημάτων των επεξεργαστών.

Προτού αναλύσουμε περαιτέρω το μοντέλο που υλοποιεί το XMT, ας δούμε την σχέση μεταξύ των μεθοδολογιών PRAM και Work-Depth. Ως μοντέλο το PRAM ορίζει πολύ αυστηρά την σχέση κώδικα-εντολών και επεξεργαστών. Ορίζει μονολιθικά για κάθε επεξεργαστή τις εντολές που εκτελεί σε κάθε χρονική στιγμή και είναι ίδιες για όλους. Ενώ το Work-Depth ως μοντέλο ορίζει πιο χαλαρά την έννοια αυτή. Δηλαδή ορίζει για κάθε επεξεργαστή, που συμμετέχει στο παράλληλο βήμα, το σύνολο των εντολών που πρέπει να εκτελέσει. Επίσημα αποδείχτηκε ότι τα δύο αυτά μοντέλα (PRAM και Work-Depth) είναι ισοδύναμα [10].

Χαλαρώνοντας τον αυστηρό περιορισμό του PRAM για συγχρονισμό των επεξεργαστών, έγινε δυνατή η υλοποίηση ενός μοντέλου όμοιου σε μεγάλο βαθμό με το PRAM μόνο με ένα μικρό κόστος εξαιτίας του ασυγχρονισμού των επεξεργαστών. Τονίζουμε ξανά ότι το μικρό κόστος αυτό είναι αμελητέο σε σχέση με το κόστος για τον πλήρη και αυστηρό συγχρονισμό των βημάτων των επεξεργαστών.

Στο υποκεφάλαιο 3.5 υπάρχει μια πιο ακριβής ανάλυση της τελικής μορφής του κώδικα της πλατφόρμα XMT αλλά και πρακτική αναφορά στο βαθμό που επηρεάζει η μεθοδολογία Work-Depth τον τελικό κώδικα.

Η ροή εκτέλεσης στο μοντέλο XMT εναλλάσσεται μεταξύ σειριακής και παράλληλης εκτέλεσης, δες Σχήμα 3.2. Όπως φαίνεται και από το σχήμα η αρχή της παράλληλης εκτέλεσης σηματοδοτείται από το spawn και τελειώνει με το join. Στο υποκεφάλαιο 3.3 θα επεξηγήσουμε πιο αναλυτικά πως συμβαίνει αυτή η εναλλαγή μεταξύ σειριακής και παράλληλης εκτέλεσης.



Σχήμα 3.2 – Ροή εκτέλεσης στο μοντέλο XMT

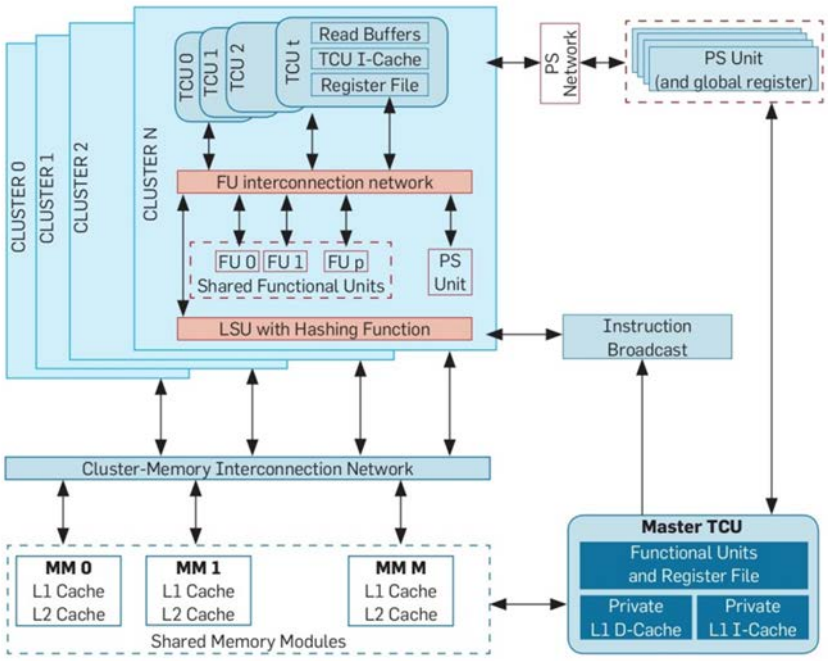
Προς το παρόν μας δίδεται η δυνατότητα να τρέξουμε κώδικα στο πρωτότυπο FPGA ή να εκτελέσουμε προσομοίωση στο cycle “accurate” προσομοιωτή. Περαιτέρω αναφορά

στο σύνολο προγραμμάτων που παρέχονται μαζί με την πλατφόρμα γίνεται στα μετέπειτα υποκεφάλαια.

3.3 Επεξεργαστής XMT

Η αρχιτεκτονική του επεξεργαστή XMT ακολουθεί σε γενικές γραμμές την διάταξη του μοντέλου παράλληλου υπολογισμού SIMD (δες Σχήμα 2.1). Στο Σχήμα 3.3 φαίνεται η διαγραμματική απεικόνιση των συνιστούντων μερών του επεξεργαστή XMT [11]. Αυτά είναι το Master Thread Control Unit (MTCU), τα processing clusters (κάθε cluster απαρτίζεται από πολλαπλά Thread Control Units - TCUs), ένα high-bandwidth low-latency δίκτυο αλληλοσύνδεσης, πολλαπλά memory modules (MM), πολλαπλά prefix sum units (PS units) και καθολικούς καταχωρητές (global registers).

Γενικά ο σχεδιασμός του επεξεργαστή XMT ακολουθεί την φιλοσοφία no-busy-wait finite-state-machines (NBW FSM), δηλαδή με πιο απλά λόγια τα συστατικά μέρη του επεξεργαστή (processors, memories, interconnection networks, etc.) δεν προκαλούν φαινόμενα busy-wait μεταξύ τους. Το μοναδικό φαινόμενο busy-wait εμφανίζεται κατά το join της παράλληλης εκτέλεσης, δηλαδή όταν συγκλίνουν οι παράλληλες εκτελέσεις στη σειριακή ροή (MTCU περιμένει τα TCUs να ολοκληρώσουν).



Σχήμα 3.3 – Διαγραμματική αναπαράσταση των μερών της αρχιτεκτονικής XMT [6]

Για να κατανοήσουμε σε γενικές γραμμές τον τρόπο λειτουργίας του επεξεργαστή ας μελετήσουμε ένα απλό σενάριο. Έστω ότι ένα σύνολο εντολών XMT καλείται να εκτελεστεί από τον επεξεργαστή XMT. Το MTCU όντας ένας εξελιγμένος σειριακός μικροεπεξεργαστής αρχίζει να εκτελεί τις εντολές σειριακά. Μια τυπική ροή της εκτέλεσης μπορεί να θεωρηθεί αυτή που φαίνεται στο Σχήμα 3.2. Όταν το MTCU συναντήσει δομή-εντολών spawn-join (δηλαδή ανιχνεύεται παράλληλη εκτέλεση), μεταδίδει το σύνολο των εντολών που εσωκλείονται σε αυτό μέσω ενός διαύλου σε όλα τα TCUs clusters και σταματά η σειριακή εκτέλεση κώδικα. Επιπρόσθετα μεταδίδονται σχετικές πληροφορίες όπως για παράδειγμα το μεγαλύτερο processor-thread ID που ορίζεται στο spawn-join block, έστω Y. Σε ένα καθολικό καταχωρητή (global register), έστω X, υπάρχει αποθηκευμένο ανά πάσα στιγμή το μεγαλύτερο thread ID από όλα τα threads που εκτελούνται ή εκτελέστηκαν. Όταν ένα TCU ολοκληρώσει την εκτέλεση ενός thread τότε εκτελώντας prefix sum (χρησιμοποιώντας το PS unit) στον καθολικό καταχωρητή X λαμβάνει το ID για το επόμενο thread που μπορεί να εκτελέσει. Εάν το ID που θα λάβει από το prefix sum είναι $\leq Y$ τότε εκτελεί το thread διαφορετικά ενημερώνει το MTCU ότι έχει ολοκληρώσει την εκτέλεση των threads. Μόλις ενημερώσουν όλα τα TCUs ότι έχουν ολοκληρώσει· τότε το MTCU συνεχίζει την σειριακή εκτέλεση του κώδικα. Εάν στην πορεία μέχρι το τέλος του XMT κώδικα υπάρχουν και άλλα spawn-join blocks τότε θα επαναληφθεί η προαναφερθείσα διαδικασία, διαφορετικά θα εκτελεστεί ο υπόλοιπος σειριακός κώδικας από το MTCU.

Σημαντικό γεγονός που πρέπει να αναφερθεί είναι ότι καθ’ όλη την διάρκεια εκτέλεσης της παράλληλης επεξεργασίας κάθε TCU είναι ανεξάρτητο από τα υπόλοιπα TCUs (θυμηθείτε NBW FSM). Επίσης αξίζει να σημειώσουμε ότι τα TCUs έχουν μια μικρή μνήμη εξαιτίας του ότι υλοποιούν τα στάδια ενός pipeline (fetch, decode, execute/mem-access, write backstages), ενώ για τις υπόλοιπες ανάγκες επεξεργασίας δεδομένων του προβλήματος (ανάγνωση δεδομένων/πληροφοριών και εγγραφή αποτελεσμάτων) χρησιμοποιείται το δίκτυο αλληλοσύνδεσης.

Η κοινόχρηστη μνήμη είναι χωρισμένη σε modules τα οποία ακολουθούν μια μορφή hashing. Συνεπώς μειώνονται τα προβλήματα συνοχής της μνήμης, τόσο σε επίπεδο bandwidth αλλά και scalability. Όπως φαίνεται και από το Σχήμα 3.3 δεν υπάρχουν local caches στα TCUs. Εντός κάθε memory module μπορούμε να πούμε ότι η σειρά εκτέλεσης πράξεων στο συγκεκριμένο εύρος μνήμης είναι προκαθορισμένη.

Για περισσότερες πληροφορίες σχετικά με την αρχιτεκτονική του XMT processor διαβάστε το [12], σχετικά με τον compiler και το runtime scheduling διαβάστε το [13] ενώ για prefetching methods διαβάστε το [14].

Μετά την γενική επεξήγηση της αρχιτεκτονικής του XMT processor είμαστε σε θέση να πούμε πιο συγκεκριμένα ότι το υλοποιημένο FPGA που παρουσιάζεται στο Σχήμα 3.1 απαρτίζεται από 64 TCUs οργανωμένα σε 4 clusters, με 16 TCUs σε κάθε cluster. Στόχος της ομάδας που αναπτύσσει το XMT είναι η δημιουργία επεξεργαστή με 1024 TCUs οργανωμένα σε 64 clusters, με 16 TCUs σε κάθε cluster.

3.4 Πλεονεκτήματα Πλατφόρμας XMT

Το μοντέλο παράλληλου υπολογισμού XMT παρουσιάζει αρκετά πλεονεκτήματα έναντι των μοντέλων υπολογισμού που εφαρμόζονται σήμερα. Μερικά από τα πλεονεκτήματα είναι τα ακόλουθα:

- Αξιοποίηση του πλούσιου παράλληλου αλγοριθμικού υποβάθρου PRAM
- Επίτευξη μέγιστου βαθμού παραλληλισμού σε κάθε βήμα (εξαιτίας PRAM)
- Είναι δυνατή η αποστολή των εντολών του προγράμματος στιγμιαία σε όλους τους επεξεργαστές (εντολών παράλληλης εκτέλεσης) σε χρόνο ίσο με όσο χρειάζεται για την ενημέρωση ενός επεξεργαστή.
- Οι επεξεργαστές που ολοκληρώνουν την εργασία τους μπορούν να χρησιμοποιηθούν για την ολοκλήρωση άλλων εργασιών. Ο χρόνος που χρειάζεται για να γίνει αυτό (δηλαδή να ανατεθούν επεξεργαστές σε νέες εργασίες) είναι ο ίδιος με αυτόν που χρειάζεται για την ανάθεση ενός μόνο επεξεργαστή σε νέα εργασία.
- Η σχετική χαμηλή συνεκτικότητα της μνήμης (CRCW, memory modules) μειώνει τα προβλήματα που παρουσιάζονται σε εύρος ζώνης διαύλου επικοινωνίας και επεκτασιμότητας [11].
- Ο επεξεργαστής XMT είναι απλός στην κατασκευή του. Συγκεκριμένα μόνο ένας απόφοιτος φοιτητής σε χρονικό διάστημα 2 χρόνων ολοκλήρωσε την περιγραφή του υλικού του πρωτότυπου επεξεργαστή XMT σε Verilog [11].
- Δεν απαιτούνται εξειδικευμένες γνώσεις για τον προγραμματισμό στην πλατφόρμα XMT. Άτομα με PRAM παιδεία είναι σε θέση να προγραμματίσουν πολύ εύκολα.

3.5 Γλώσσα Προγραμματισμού XMTC

Η γλώσσα προγραμματισμού XMTC μας δίνει την δυνατότητα να προγραμματίσουμε στην πλατφόρμα XMT. Αποτελεί μία προέκταση της γλώσσας προγραμματισμού C με κάποια επιπλέον στοιχεία. Εξαιτίας του ότι η πλατφόρμα XMT βρίσκεται υπό ανάπτυξη δεν υποστηρίζονται όλες οι δυνατότητες που παρέχονται από την γλώσσα προγραμματισμού C. Αναφορά στους περιορισμούς που γίνεται στα μετέπειτα σχετικά υποκεφάλαια. Όπως τονίζεται από την ομάδα που αναπτύσσει την πλατφόρμα XMT στο μέλλον θα προσπαθήσουν διευρύνουν τις δυνατότητες της πλατφόρμας ούτως ώστε να καλύπτει περισσότερες προγραμματιστικές ανάγκες.

3.5.1 Από το PRAM στην XMTC

Προκειμένου να κατανοήσουμε την αντιστοιχία μεταξύ του PRAM κώδικα και XMTC κώδικα θα παραθέσουμε ένα απλό παράδειγμα. Θυμηθείτε πώς η ροή στην πλατφόρμα XMT ακολουθεί εναλλαγές μεταξύ σειριακής και παράλληλης εκτέλεσης (δες Σχήμα 3.2 στο υποκεφάλαιο 3.2).

Έστω ότι επιθυμούμε να επιλύσουμε ένα πρόβλημα άθροισης πινάκων. Το ζητούμενο είναι τα αθροίσουμε τα αντίστοιχα στοιχεία των πινάκων A και B, και το άθροισμα που προκύπτει να αποθηκευτεί στην αντίστοιχη θέση στον πίνακα C. Το μέγεθος των πινάκων είναι m. Στον ψευδοκώδικα PRAM (Ψευδοκώδικας 3.1) και κώδικα XMTC

```
PRAM Pseudo-Code

LN
01    Processors i=0 to m-1 do in parallel:
02    C[i]:=A[i]+B[i]
03    End do
```

Ψευδοκώδικας 3.1 – Άθροισμα Πινάκων (μοντέλο PRAM)

(Κώδικας XMTC 3.1) φαίνεται εν συντομία (για λόγο απλούστευσης) η προτεινόμενη λύση στο μοντέλο PRAM και στο XMT αντίστοιχα. Προσέξτε ότι ο κώδικας XMTC είναι σχεδόν όμοιος με τον αντίστοιχο PRAM ψευδοκώδικα. Σημαντικά σημεία που

```
XMTC Code

LN
01    spawn(0, m-1){
02          C[$] = A[$] + B[$];
03    }
```

Κώδικας XMTC 3.1 – Άθροισμα Πινάκων

πρέπει να προσέξουμε είναι ότι και στις δύο περιπτώσεις δημιουργούνται m virtual threads τα οποία τρέχουν παράλληλα. Η δομή “Processors … do in parallel” στον PRAM ψευδοκώδικα μετατρέπεται σε δομή spawn-join στον XMTC κώδικα. Επίσης αξιοσημείωτο είναι το γεγονός ότι στον PRAM ψευδοκώδικα το ID του κάθε virtual thread αναπαρίσταται με τον χαρακτήρα `i`, ενώ στον κώδικα XMTC χρησιμοποιείται ο χαρακτήρας `$`.

Σε αυτό το σημείο κανονικά πρέπει να αναφερθούμε και στην διαφορά μεταξύ PRAM και XMTC εξαιτίας της μεθοδολογίας Work-Depth αλλά αυτό είναι αντικείμενο σχολιασμού στο υποκεφάλαιο 3.5.8 (ο λόγος για τον οποίο γίνεται αυτό είναι γιατί πρέπει πρώτα να επεξηγηθούν οι κανόνες σύνταξης και οι εντολές της XMTC).

3.5.2 Βιβλιοθήκες

Απαραίτητη προϋπόθεση για την συγγραφή κώδικα XMTC είναι η βιβλιοθήκη `<xmtc.h>`. Επίσης είναι δυνατή η χρήση βιβλιοθηκών ορισμένες από τον χρήστη-προγραμματιστή αλλά με την προϋπόθεση να λαμβάνονται υπόψη οι περιορισμοί της XMTC (όπως αυτοί περιγράφονται στα manuals [15] [16]).

3.5.3 Δομή Spawn

Αυτή η δομή επιτρέπει την εκτέλεση παράλληλου κώδικα στην πλατφόρμα XMT. Πιο συγκεκριμένα επιτρέπει την δημιουργία πλήθους (end_thread – start_thread + 1) virtual threads οι οποίες εκτελούνται παράλληλα. Τα threads αποκτούν από ένα μοναδικό αναγνωριστικό (thread ID) το οποίο ανήκει στο σύνολο [start_thread, end_thread] συμπεριλαμβανομένων. Εντός της δομής είναι δυνατή η χρήση του thread ID με την χρήση του χαρακτήρα `$`. Κατά την διάρκεια εκτέλεσης των threads λέμε ότι ο επεξεργαστής βρίσκετε στο parallel mode of execution.

```
spawn(start_thread, end_thread){
   //Spawn Block :
   // insert parallel thread code here
}
```

Επίσης λόγω του ότι η πλατφόρμα XMT βρίσκεται υπό ανάπτυξη υπάρχουν περιορισμοί που πρέπει να λαμβάνονται υπόψη. Αξίζει να αναφέρουμε μερικούς βασικούς περιορισμούς, όπως για παράδειγμα το μέγιστο αριθμό εντολών (assembly

instructions) που μπορούν να εσωκλείονται εντός του spawn block ο οποίος είναι 1000 εντολές. Επίσης εάν το πλήθος τον virtual threads ξεπεράσει το όριο των 64 για το FPGA και 1024 για τον προσομοιωτή, τότε η πλατφόρμα σε κάθε περίπτωση αρχίζει να μεταχειρίζεται τα επιπλέον virtual threads με Round-Robin μεθοδολογία (δηλαδή δεν γίνεται ταυτόχρονη μεταχείριση, για περισσότερες πληροφορίες σχετικά με τον τρόπο εκτέλεσης δες υποκεφάλαιο 3.3). Για πιο αναλυτική περιγραφή των περιορισμών που υφίσταται η δομή ανατρέξτε στο εγχειρίδιο χρήσης [15].

3.5.4 Δομή Single Spawn

Με την δομή αυτή γίνεται δυνατόν να δημιουργηθούν επιπλέον virtual threads κατά την εκτέλεση κάποιου virtual thread. Συντακτικά η δομή sspawn μπορεί να δηλωθεί εντός ενός spawn block.

```
spawn(start_thread, end_thread){
  int child_ID;

  //Some parallel code here

  sspawn(child_ID){
    //Initialization Block:
    //Code for initializing the child thread
  }

  //Some other parallel code here
}
```

Για να κατανοήσουμε καλύτερα την χρησιμότητα της δομής sspawn ας εξηγήσουμε τι συμβαίνει σε περίπτωση που υπάρχει δομή sspawn εντός spawn block. Το virtual thread πατέρας εκτελεί κανονικά τις εντολές που βρίσκονται εντός του spawn block, από την πρώτη εντολή μέχρι την τελευταία. Μόλις ο πατέρας συναντήσει δομή sspawn τότε δημιουργεί ένα επιπλέον virtual thread (child) με τιμή αναγνωριστικού (child_ID). Ο πατέρας συνεχίζει την εκτέλεση των εντολών που βρίσκονται στο initiation block. Αντιθέτως το virtual thread (child) που δημιουργείται αρχίζει να εκτελεί εντολές από την αρχή του spawn block του πατέρα. Εδώ εμφανίζεται ένα race condition μεταξύ παιδιού και πατέρα. Επίσης πρέπει να λάβει κανείς υπόψη του θέματα συγχρονισμού αλλά και ειδική μεταχείριση για την αποφυγή δημιουργίας απεριόριστου αριθμού virtual threads (infinite spawn of virtual threads). Δεν θα αναφέρουμε επιπλέον

λεπτομέρειες διότι η συγκεκριμένη δομή δεν χρησιμοποιήθηκε κατά την ολοκλήρωση της παρούσας Ατομικής Διπλωματικής Εργασίας. Ο λόγος για τον οποίο γίνεται αναφορά στην δομή single spawn (sspawn) είναι διότι επιθυμούμε να τονίσουμε τις δυνατότητες της XMTC. Για περισσότερες λεπτομέρειες αναφορικά με την δομή single spawn και τους περιορισμούς που την διέπουν ανατρέξτε στο εγχειρίδιο χρήσης [15].

3.5.5 Εντολές Prefix Sum και Prefix Sum to Memory

Η γενική μορφή των εντολών προθεματικού αθροίσματος ps και psm φαίνεται πιο κάτω. Και οι δύο εντολές εκτελούν τις ακόλουθες διεργασίες:

- Προσθέτει την τιμή local_integer στην δεύτερη παράμετρο (ps_base για την εντολή ps, διεύθυνση μνήμης για psm).
- Αντιγράφει την παλιά τιμή της δεύτερης παραμέτρου στην πρώτη παράμετρο local_integer.

```
ps(int local_integer, psBaseReg ps_base);
psm(int local_integer, int variable);
```

Όσον αφορά την αποδοτικότητα των δύο εντολών, πρέπει να πούμε ότι η εντολή ps είναι πιο αποδοτική από την psm εξαιτίας του ότι είναι δυνατή η εκτέλεση της ps από πολλαπλά virtual threads στην ίδια μεταβλητή psBaseReg ταυτόχρονα. Αντιθέτως η εκτέλεση της psm στην ίδια θέση μνήμης από πολλαπλά virtual threads την ίδια στιγμή θα οδηγήσει σε σειριακή ολοκλήρωση της διαδικασίας στο σύνολο της. Ο λόγος για τον οποίο συμβαίνει αυτό είναι εν μέρει λόγο αρχιτεκτονικής, αφού οι μεταβλητές psBaseReg είναι registers ενσωματωμένοι στην αρχιτεκτονική με καλύτερη πρόσβαση ενώ οι διευθύνσεις μνήμης βρίσκονται σε σημεία της μνήμης που είναι πιο δύσκολη η πρόσβαση σε αυτά. Για περισσότερες λεπτομέρειες σχετικά με τις εντολές ps και psm, αλλά και τους περιορισμούς παρακαλείστε όπως ανατρέξτε στο εγχειρίδιο χρήσης [15].

3.5.6 Μεταβλητές

Η γλώσσα XMTC ακολουθεί τους ίδιους κανόνες για το scope των μεταβλητών (τοπικών και καθολικών). Προσοχή πρέπει να δοθεί στην ιδιαιτερότητα της XMTC ότι οι μεταβλητές που ορίζονται εντός spawn block (spawn) και single spawn block

(sspawn) είναι προσωπικές για κάθε virtual thread. Επίσης χρειάζεται προσοχή στην μεταχείριση των καθολικών μεταβλητών και γενικότερα των θέσεων μνήμης που βρίσκονται στην κοινόχρηστη μνήμη εξαιτίας του γεγονότος ότι μπορεί οποιοδήποτε virtual thread να μεταβάλει τις τιμές.

Οι τύποι δεδομένων που υποστηρίζονται είναι integers και ο ειδικός τύπος psBaseReg για εκτέλεση προθεματικών αθροισμάτων (prefix sum). Αυτοί οι δύο τύποι θεωρούνται αξιόπιστοι για χρήση. Ακόμη υπάρχει και ο τύπος float απλώς βρίσκεται σε δοκιμαστικό στάδιο. Βάση της ομάδας που αναπτύσσει την πλατφόρμα τονίζεται το γεγονός ότι στο μέλλον σκοπεύουν να εντάξουν περισσότερους τύπους δεδομένων. Για περισσότερες λεπτομέρειες σχετικά με τις μεταβλητές αλλά και τους περιορισμούς παρακαλείστε όπως ανατρέξτε στο εγχειρίδιο χρήσης [15].

3.5.7 Συναρτήσεις και Κλίσεις Συστήματος

Καταρχάς πρέπει να τονίσουμε τον βασικό περιορισμό ότι εντός των spawn blocks δεν επιτρέπεται η κλίση σε οποιανδήποτε συνάρτηση. Προς το παρόν υποστηρίζεται η κλίση συναρτήσεων οι οποίες ορίζονται από τον προγραμματιστή (user defined functions) και οι οποίες βρίσκονται εκτός spawn blocks. Επίσης προς το παρόν δεν είναι δυνατή η εκτέλεση κλίσεων συστήματος (system calls) όπως για παράδειγμα εκτελέσεις I/O, συναρτήσεις από την βιβλιοθήκη math και άλλα. Όσον αφορά την είσοδο δεδομένων στο πρόγραμμα είναι δυνατή η χρήση του εργαλείου Memory Tool που παρέχεται μαζί με την πλατφόρμα XMT. Για την εξαγωγή δεδομένων από το πρόγραμμα υποστηρίζεται κλίση printf όμως με κάποιους περιορισμούς. Για περισσότερες λεπτομέρειες σχετικά με τις συναρτήσεις και κλίσεις συστήματος αλλά και τους περιορισμούς παρακαλείστε όπως ανατρέξτε στο εγχειρίδιο χρήσης [15].

3.5.8 Μεθοδολογία Work-Depth και XMTC

Σε αυτό το σημείο είμαστε σε θέση να εξηγήσουμε ακριβώς το πώς δένει η μεθοδολογία Work-Depth με την γλώσσα XMTC. Αξίζει να θυμίσουμε το μοντέλο ροής εκτέλεσης της πλατφόρμας XMT (δες υποκεφάλαιο 3.2 και 3.3), βάση του οποίου κατά την σειριακή εκτέλεση του προγράμματος είναι δυνατή η δημιουργία πολλαπλών threads τα οποία μπορούν να εκτελούνται ταυτοχρόνως. Κατά την φάση της παράλληλης λειτουργίας κάθε thread τρέχει με την δική του ταχύτητα, δηλαδή τα βήματα των επεξεργαστών είναι ασυγχρόνιστα. Θυμηθείτε ότι η πλατφόρμα XMT είναι

υλοποίηση μιας πιο χαλαρής έκδοσης του μοντέλου PRAM (όπως αναφέραμε υποκεφάλαιο 3.2) στην οποία τα βήματα των επεξεργαστών δεν είναι συγχρονισμένα. Λογικό είναι να διερωτηθεί κανείς για το πώς γίνεται ένας κώδικας με βάση το μοντέλο PRAM (όπου τα βήματα των επεξεργαστών-threads είναι συγχρονισμένα) να μπορεί να εκτελείται στην πλατφόρμα XMT (στη οποία τα βήματα των threads δεν είναι συγχρονισμένα). Η απάντηση στο ερώτημα αυτό είναι ότι η μεθοδολογία Work-Depth έρχεται να καλύψει αυτό το κενό. Πάντα η μελέτη ενός παραδείγματος βοηθάει στην κατανόηση νέων εννοιών, επομένως ας μελετήσουμε το πρόβλημα της παράλληλης άθροισης.

Καταρχήν το πρόβλημα της παράλληλης άθροισης ορίζει ότι δεδομένου ενός πίνακα τιμών πρέπει να υπολογίσουμε το άθροισμά τους (για περισσότερες λεπτομέρειες δες Κεφάλαιο 4). Ο ψευδοκώδικας PRAM για την λύση του προβλήματος στο αυστηρά ορισμένο μοντέλο PRAM φαίνεται πιο κάτω (Ψευδοκώδικας 3.2). Για τον ψευδοκώδικα PRAM που παρουσιάσαμε μπορούμε να κάνουμε κάποιες παρατηρήσεις. Πρώτον ο ψευδοκώδικας αυτός περιέχει το σύνολο των εντολών που εκτελεί ο κάθε επεξεργαστής – thread. Δεύτερον κατά οποιαδήποτε τυχαία χρονική στιγμή όλοι οι επεξεργαστές – threads εκτελούν την ίδια εντολή, αφού είναι συγχρονισμένα τα βήματα.

```
PRAM Pseudo-Code

LN
01    Processors i=1 to n/2 do in parallel:
02    Shared array sum[1..n]
03    Private j,a,b
04    Set j=1
05    While (i<=n/2j)
06          a = sum[2i-1]
07          b = sum[2i]
08          sum[i] = a + b
09          j = j+1
10    End while
11    End do
```

Ψευδοκώδικας 3.2 – Παράλληλη άθροιση (μοντέλο PRAM)

Στο μοντέλο Work-Depth είναι δυνατή η επίτευξη ενός ισοδύναμου αποτελέσματος με το μοντέλο PRAM (είναι ισοδύναμα μοντέλα και τα δύο [10]). Επίσης είναι δυνατή η μετατροπή οποιουδήποτε αλγορίθμου PRAM σε Work-Depth και το αντίστροφο. Η διαφορά των δύο μοντέλων είναι ότι στο μοντέλο Work-Depth ο κώδικας δεν είναι

μονολιθικός για τον κάθε επεξεργαστή-thread. Με τον όρο μονολιθικός εννοούμε ότι ο κάθε επεξεργαστής-thread δεν έχει στην διάθεση του ολόκληρο τον κώδικα που επιλύει το ζητούμενο. Δηλαδή σε κάθε βήμα, ο κάθε επεξεργαστής εκτελεί τις εντολές που ολοκληρώνουν το συγκεκριμένο βήμα. Ο ψευδοκώδικας για το μοντέλο Work-Depth φαίνεται πιο κάτω (Ψευδοκώδικας 3.3). Παρατηρείστε ότι οι επεξεργαστές-threads εκτελούν δουλειά μόνο στα πλαίσια του κάθε βήματος.

```
Work-Depth Pseudo-Code (sync. processors)

LN
01    i=n/2
02    While (i>=1) do
03      Processors i=1 to n/2 do in parallel:
04        Shared array sum[1..n]
05        Private a,b
06        a = sum[2i-1]
07        b = sum[2i]
08        sum[i] = a + b
09      End do
10      i=i/2
11    End do
```

Ψευδοκώδικας 3.3 – Παράλληλη άθροιση (μοντέλο Work-Depth)

Το δυνατό σημείο του μοντέλου αυτού είναι ότι με μια μικρή μετατροπή στο κώδικα είναι δυνατόν να πετύχουμε ορθότητα του αποτελέσματος ανεξάρτητα του εάν οι επεξεργαστές είναι συγχρονισμένοι ή ασυγχρόνιστοι. Πιο κάτω φαίνεται ο ψευδοκώδικας που πετυχαίνει αυτό το ζητούμενο (Ψευδοκώδικας 3.4). Μάλιστα ο

```
Work-Depth Pseudo-Code (async. processors)

LN
01    r=1
02    i=n/2
03    While (i>=1) do
04      Processors i=1 to n/2 do in parallel:
05        Shared array sum[logn][1..n]
06        Private a,b
07        a = sum[r-1][2i-1]
08        b = sum[r-1][2i]
09        sum[r][i] = a + b
10      End do
11      r=r+1
12      i=i/2
13    End do
```

Ψευδοκώδικας 3.4 – Παράλληλη άθροιση (μοντέλο asynchronous WD)

ψευδοκώδικας μπορεί να μετατραπεί σε κώδικα XMTC πολύ εύκολα (αντικατάσταση της δομής παράλληλης εκτέλεσης με δομή spawn).

Έχοντας υπόψη τα δύο μοντέλα PRAM και Work-Depth, πλέον κατανοούμε γιατί το μοντέλο Work-Depth συσχετίζεται σε τόσο μεγάλο βαθμό με την πλατφόρμα XMT. Ο συσχετισμός των μοντέλων αυτών στηρίζεται μεν σε μικρές λεπτομέρειες αλλά στην πράξη ήταν τόσο σημαντικές ώστε οδήγησαν στην υλοποίηση της πλατφόρμας XMT.

Για την υλοποίηση οποιουδήποτε αλγορίθμου PRAM στην πλατφόρμα XMT, αρκεί κανείς να χρησιμοποιήσει την μεθοδολογία Work-Depth για να μετατρέψει τον αλγόριθμο PRAM. Η μεθοδολογία Work-Depth με απλά λόγια ορίζει ότι αρκεί να εντοπίσει κανείς την δομή (συνήθως είναι δομή επανάληψης) που επιτρέπει την εναλλαγή των βημάτων και να την αφαιρέσει εκτός του παράλληλου κομματιού.

3.6 Εργαλείο Μορφοποίησης Μνήμης

Το εργαλείο μορφοποίησης μνήμης ή αλλιώς Memory Tool, παρέχεται μαζί με την πλατφόρμα XMT. Το εργαλείο αυτό μας δίδει την δυνατότητα να προετοιμάσουμε την μνήμη (μεταβλητές, πίνακες, κ.α.) πριν την εκτέλεση του προγράμματος. Το εργαλείο αυτό υπάρχει ως υποκατάστατο της μη υποστήριξης κλίσεων συστήματος για την εισαγωγή δεδομένων σε προγράμματα. Στο μέλλον με την υποστήριξη κλίσεων συστήματος I/O το παρόν εργαλείο θα καταργηθεί.

Με την χρήση του εργαλείου Memory Tool παράγονται τα εξής αρχεία:

- Ένα αρχείο header (.h) το οποίο περιέχει τις δηλώσεις των δομών της μνήμης (μεταβλητές, πίνακες, κ.α.) οι οποίες θα συμπεριληφθούν μαζί με τον κώδικα XMTC.
- Ένα binary αρχείο (.xbo) το οποίο θα τροφοδοτηθεί στον compiler. Το αρχείο αυτό περιέχει τις αρχικοποιημένες τιμές της μνήμης.
- Ένα απλό αρχείο κειμένου (.txt) στο οποίο φαίνονται τα περιεχόμενα του binary αρχείου που προαναφέραμε. Ο σκοπός του παρόν αρχείου είναι καθαρά για να βοηθήσει στο debugging του προγράμματος.

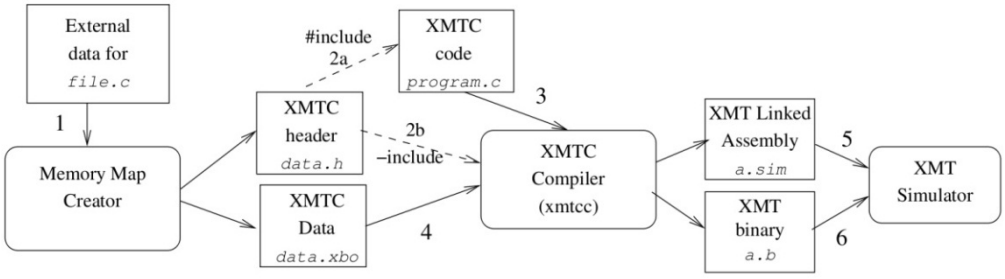
Για περισσότερες πληροφορίες σχετικά με την χρήση του εργαλείου αλλά και τους περιορισμούς του παρακαλείστε όπως ανατρέξτε στο εγχειρίδιο χρήσης [15].

3.7 Εργαλείο Σειριοποίησης

Το εργαλείο σειριοποίησης (XMTC Serializer) μετατρέπει τον παράλληλο κώδικα XMTC σε σειριακό. Ο λόγος για τον οποίο μας παρέχεται αυτή η δυνατότητα είναι γιατί στο παρόν στάδιο ανάπτυξης της πλατφόρμας XMT δεν υπάρχει τρόπος για αποσφαλμάτωση και έλεγχο του παράλληλου κώδικα που γράφουμε. Έτσι χρησιμοποιώντας τον XMTC Serializer παίρνουμε την σειριακή εκδοχή του παράλληλου κώδικα XMTC, η οποία μπορεί να μεταγλωττιστεί σε ένα κλασσικό compiler όπως για παράδειγμα ο gcc. Να τονίσουμε ότι κατά την χρήση του εργαλείου XMTC Serializer δεν καταστρέφεται ο αρχικός κώδικας XMTC με τον οποίο τροφοδοτείται το εργαλείο. Για περισσότερες πληροφορίες σχετικά με την χρήση του εργαλείου αλλά και τους περιορισμούς του παρακαλείστε όπως ανατρέξτε στο εγχειρίδιο χρήσης [15].

3.8 Μεταγλώττιση και Προσομοίωση Προγράμματος XMTC

Στο Σχήμα 3.4 παρουσιάζονται οι διαδικασίες μορφοποίησης της μνήμης (Memory Tool), μεταγλώττισης του XMTC κώδικα και τέλος η διαδικασία προσομοίωσης (XMT Simulator). Στο παρόν υποκεφάλαιο θα ασχοληθούμε με τις διαδικασίες μεταγλώττισης και προσομοίωσης. Αναφορά στην διαδικασία μορφοποίησης της μνήμης γίνεται στο υποκεφάλαιο 3.6.



Σχήμα 3.4 – Ροή ανάπτυξης και προσομοίωσης προγράμματος στην πλατφόρμα XMT [10]: 1. Προετοιμασία μνήμης χρησιμοποιώντας το Memory Tool, 2a. Συμπερίληψη header file στον κώδικα με `#include`, 2b.Συμπερίληψη header file με παράμετρο `–include` κατά την κλίση του compiler, 3. Μεταγλώττιση του κώδικα XMTC, 4. Ανάγνωση δυαδικού αρχείου data.xbo από τον compiler, 5. Προσομοίωση του XMT assembly κώδικα, 6. Φόρτωση δεδομένων μνήμης στον προσομοιωτή (με χρήση της παραμέτρου `–binload`)

Στα επόμενα υποκεφάλαια παρουσιάζεται συνοπτικά η διαδικασία μεταγλώττισης και προσομοίωσης. Για περισσότερες πληροφορίες σχετικά με την χρήση των εργαλείων

μεταγλώττισης και προσομοίωσης αλλά και τους περιορισμούς τους παρακαλείστε όπως ανατρέξτε στο εγχειρίδιο χρήσης [15].

3.8.1 Μεταγλώττιση Κώδικα XMTC

Για την μεταγλώττιση κώδικα XMTC πρέπει κανείς να καλέσει τον μεταγλωττιστή με την εντολή `xmtcc`. Όπως φαίνεται από το Σχήμα 3.4 αλλά και όπως πιθανών διαπιστώσατε υπάρχουν δύο πιθανά ενδεχόμενα. Στην πρώτη περίπτωση το πρόγραμμα που επιθυμούμε να μεταγλωττίσουμε κάνει χρήση του εργαλείου μορφοποίησης μνήμης (Memory Tool) ενώ στην δεύτερη περίπτωση το πρόγραμμα δεν κάνει χρήση του εργαλείου μορφοποίησης μνήμης.

Για την πρώτη περίπτωση ο χρήστης – προγραμματιστής πρέπει να καλέσει τον μεταγλωττιστή με την εξής εντολή:

```
Terminal$ xmtcc program.c –include data.h data.xbo
```

Στην δεύτερη περίπτωση που ο χρήστης – προγραμματιστής δεν κάνει χρήση του εργαλείου μορφοποίησης μνήμης, πρέπει να καλέσει τον μεταγλωττιστή με την εξής εντολή:

```
Terminal$ xmtcc program.c
```

3.8.2 Προσομοίωση του XMT Assembly Κώδικα

Για την προσομοίωση του XMT assembly κώδικα που προκύπτει από την μεταγλώττιση πρέπει κανείς να καλέσει τον προσομοιωτή με την εντολή `xmtsim`. Στο σημείο αυτό χρειάζεται να επισημάνουμε μια ιδιαιτερότητα που πρέπει να έχει υπόψη του ο χρήστης – προγραμματιστής. Ο προσομοιωτής έχει τους εξής δύο τρόπους για την ολοκλήρωση προσομοιώσεων:

- Assembly Simulation: αυτός ο τρόπος εκτελεί σειριακή προσομοίωση των thread, ξεκινώντας πρώτα με το thread με το μικρότερο ID. Αυτός ο τρόπος είναι γρηγορότερος σε εκτέλεση αλλά δεν ενδείκνυται για ρεαλιστική προσομοίωση. Κυρίως χρησιμοποιείται για debugging.
- Cycle-accurate Simulation: είναι ο πιο ρεαλιστικός τρόπος προσομοίωσης, αφού εκτελείται παράλληλη εκτέλεση των threads. Είναι ότι πιο πλησιέστερο στην

πλατφόρμα XMT. Επίσης με αυτό τον τρόπο είναι δυνατή η εξαγωγή συνολικού αριθμού κύκλων μέχρι την ολοκλήρωση της προσομοίωσης.

Για να προσομοιώσει κανείς με την πρώτη μέθοδο (assembly simulation) πρέπει να εκτελέσει την ακόλουθη εντολή (έχοντας υπόψη το Σχήμα 3.4):

```
Terminal$ xmtsim –binload a.b a.sim
```

Εάν ο χρήστης – προγραμματιστής επιθυμεί να προσομοιώσει με την δεύτερη μέθοδο (cycle-accurate simulation) πρέπει να εκτελέσει την ακόλουθη εντολή (λαμβάνοντας υπόψη το Σχήμα 3.4):

```
Terminal$ xmtsim –cycle –binload a.b a.sim
```

3.9 Εργαλείο XMTC-To-OpenMP

Μαζί με την πλατφόρμα XMT υπάρχει και το εργαλείο XMTC-To-OpenMP το οποίο μετατρέπει τον XMTC κώδικα σε C, ο οποίος κώδικας C περιέχει OpenMP directives. Ο λόγος για τον οποίο υπάρχει αυτό το εργαλείο είναι για να μας παρέχει την δυνατότητα να εκτελούμε το πρόγραμμα σε πολυπύρηνες πλατφόρμες (multicore platforms). Για περισσότερες πληροφορίες σχετικά με την χρήση του εργαλείων μεταγλώττισης και προσομοίωσης αλλά και τους περιορισμούς του παρακαλείστε όπως ανατρέξτε στο εγχειρίδιο χρήσης [15].

3.10 Υπολογιστικό Σύστημα Πλατφόρμας

Όλες οι υλοποιήσεις και αξιολογήσεις των αλγορίθμων έγιναν στην πλατφόρμα XMT, η οποία εγκαταστάθηκε σε υπολογιστικό σύστημα με τα εξής χαρακτηριστικά:

```
OS: Debian GNU/Linux 6.0.4 (squeeze)
Kernel: 2.6.32-5-686-bigmem
Processor: Intel(R) Core(TM)2 Quad CPU    Q9650  @ 3.00GHz
Memory: 4.0 GiB
```

Κεφάλαιο 4

Παράλληλη Άθροιση

4.1 Πρόβλημα Άθροισης

Το πρόβλημα της άθροισης ορίζεται ως εξής: Δεδομένου ενός συνόλου αριθμών A το οποίο αποτελείται από n αριθμούς $A = \{\alpha_1, \alpha_2, \dots, \alpha_n\}$, πρέπει να υπολογίσουμε το άθροισμα S των αριθμών (δηλαδή $S = \sum_{i=1}^{n} \alpha_i$).

4.2 Αλγόριθμος Σειριακής Άθροισης

Ο αλγόριθμος που μας παρέχει τον σειριακό τρόπο λύσης του προβλήματος φαίνεται πιο κάτω (Ψευδοκώδικας 4.1). Αξίζει να αναφέρουμε ότι στην σειριακή επίλυση οποιουδήποτε προβλήματος πάντα γίνεται χρήση μόνο ενός επεξεργαστή.

Αναλύοντας τον σειριακό τρόπο λύσης παρατηρούμε ότι ο επεξεργαστής χρειάζεται n επαναλήψεις προκειμένου να αθροίσει όλους τους αριθμούς.

```
Set S := a_1

For i=2 to n do
     S := S + a_i
end do
```

Ψευδοκώδικας 4.1 – Σειριακή άθροιση

Σημαντικός παράγοντας για οποιοδήποτε αλγόριθμο είναι η πολυπλοκότητα του. Ο σειριακός αλγόριθμος άθροισης χρησιμοποιεί μόνο ένα επεξεργαστή και χρειάζεται n προσβάσεις στην μνήμη για να διαβάσει όλους τους αριθμούς. Ο συνολικός χρόνος για την επίλυση του προβλήματος από τον επεξεργαστή είναι $\Theta(n)$. Για να είμαστε σε θέση να συγκρίνουμε εύκολα τον σειριακό με τον παράλληλο αλγόριθμο πιο μετά θα ορίσουμε την πολυπλοκότητα του σειριακού αλγόριθμου χρησιμοποιώντας τις μετρικές που παρουσιάσαμε στο υποκεφάλαιο 2.2.

Πλήθος επεξεργαστών: $\mathrm{P}(n) = \rho = 1$

Χρόνος ολοκλήρωσης σειριακού αλγορίθμου: $\mathrm{T}_1(n) = \Theta(n)$

Κόστος σειριακού αλγορίθμου: $\mathrm{C}_1(n) = \mathrm{T}_1(n) \times \mathrm{P}(n) = \Theta(n)$

Παρατηρούμε ότι το κόστος του αλγορίθμου ισούται με το χρόνο ολοκλήρωσης. Αυτό που επιθυμούμε να επιτύχουμε χρησιμοποιώντας τον παράλληλο αλγόριθμο άθροισης είναι να μειώσουμε τον χρόνο εκτέλεσης και ταυτοχρόνως να μην αυξήσουμε σημαντικά το κόστος του.

4.3 Αλγόριθμος Παράλληλης Άθροισης

Ο αλγόριθμος παράλληλης άθροισης χρησιμοποιώντας δύο ή περισσότερους επεξεργαστές προσπαθεί να επιλύσει το πρόβλημα της άθροισης (δες υποκεφάλαιο 4.1) σε χρόνο μικρότερο από αυτόν που χρειάζεται ο καλύτερος σειριακός αλγόριθμος άθροισης. Επίσης φροντίζει ώστε το κόστος του να μη αυξηθεί σημαντικά.

4.3.1 Περιγραφή Αλγ. Παράλληλης Άθροισης

Ο αλγόριθμος παράλληλης άθροισης προσπαθεί να επιλύσει ένα πρόβλημα μεγέθους n, απλοποιώντας το σε μικρότερα ιεραρχημένα προβλήματα. Χρησιμοποιώντας πολλαπλά

βήματα επιτυγχάνει την πλήρη επίλυση του προβλήματος. Πιο συγκεκριμένα σε κάθε του βήμα ο αλγόριθμος παράλληλης άθροισης έχει $(n/2)$ ενεργούς επεξεργαστές, όπου n το πλήθος των στοιχείων που πρέπει να αθροιστούν. Με την εκτέλεση διαδοχικών βημάτων όπως διαφαίνεται το πλήθος των ενεργών επεξεργαστών και των στοιχείων που πρέπει να αθροιστούν μειώνεται στο μισό (πλήθος ενεργών επεξεργαστών ≠ πλήθος των στοιχείων που πρέπει να αθροιστούν). Η γενική μορφή του αλγορίθμου παράλληλης άθροισης παρουσιάζεται πιο κάτω (Ψευδοκώδικας 3.1).

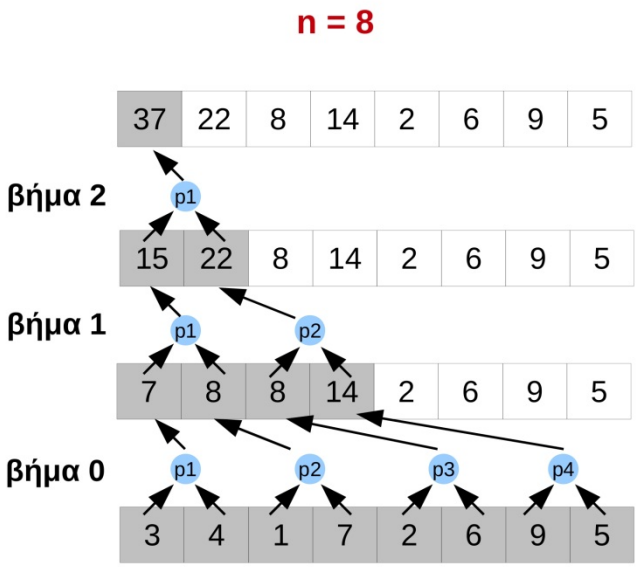
```
PRAM Pseudo-Code

LN
01   Processors i=1 to n/2 do in parallel:
02   Shared array sum[1..n]
03   Private j,a,b
04   Set j=1
05   While (i<=n/2j)
06        a = sum[2i-1]
07        b = sum[2i]
08        sum[i] = a + b
09        j = j+1
10   End while
11   End do
```

Ψευδοκώδικας 4.2 – Παράλληλη άθροιση

Για την καλύτερη κατανόηση του αλγορίθμου παράλληλης άθροισης θα αναλύσουμε σε κάποιο βαθμό το σκεπτικό που υπάρχει πίσω από αυτόν. Γενικά ο οποιοσδήποτε αλγόριθμος που επιλύει κάποιο πρόβλημα περιστρέφεται γύρω από δύο άξονες. Ο πρώτος άξονας είναι η αναπαράσταση των δεδομένων στην μνήμη (memory mapping). Ο δεύτερος άξονας είναι το πώς γίνεται η επεξεργασία αυτών των δεδομένων (data processing). Ο λόγος για τον οποίο γίνεται αναφορά στους άξονες είναι για την επίτευξη ενός από τους σκοπούς της παρούσας Ατομικής Διπλωματικής Εργασίας (δες υποκεφάλαιο 1.2).

Έστω ότι το πρόβλημα άθροισης που επιθυμούμε να επιλύσουμε είναι μεγέθους $(n = 8)$. Όπως φαίνεται στο Σχήμα 4.1 αρχικά τα δεδομένα βρίσκονται σε ένα πίνακα n θέσεων. Έχοντας υπόψη και τον ψευδοκώδικα (Ψευδοκώδικας 4.2), παρατηρείστε ότι κατά το πρώτο βήμα του αλγορίθμου ‘βήμα 0’ υπάρχουν $(n/2)$ ενεργοί επεξεργαστές,

οι οποίοι αναλαμβάνουν ο καθένας ξεχωριστά να αθροίσει από δύο διαδοχικά στοιχεία. Αυτή η διαδικασία περιλαμβάνει δύο αναγνώσεις από την κοινόχρηστη μνήμη (lines 06-07), υπολογισμό αθροίσματος των δύο αριθμών και αποθήκευση τους στην κοινόχρηστη μνήμη (line 08). Οι προαναφερθείσες πράξεις λαμβάνουν χώρα για κάθε ενεργό επεξεργαστή ξεχωριστά. Θυμηθείτε από το υποκεφάλαιο 2.4.1 ότι οι επεξεργαστές στο μοντέλο PRAM είναι συγχρονισμένοι, επομένως δεν δημιουργείται οποιοδήποτε θέμα ασυνέπειας δεδομένων/πληροφοριών στην κοινόχρηστη μνήμη κατά



Σχήμα 4.1 – Παράλληλη άθροιση ($n = 8$)

τις εγγραφές ή αναγνώσεις. Ο βρόγχος που ορίζεται στον αλγόριθμο (lines 05-10) διασφαλίζει ότι ο αριθμός των επεξεργαστών που απομένουν ενεργοί μετά το πέρας κάθε βήματος είναι ο μισός. Ο συγκεκριμένος βρόγχος ορίζει το πλήθος των βημάτων που θα εκτελεστούν. Κατά το τελευταίο βήμα, 'βήμα 2', υπάρχει μόνο ένας επεξεργαστής ενεργός ο οποίος αθροίζει τα δύο υπό-αθροίσματα και αποθηκεύει το συνολικό άθροισμα στην κοινόχρηστη μνήμη. Παρατηρείστε ότι σε κάθε βήμα οι επεξεργαστές αποθηκεύουν το άθροισμα που υπολογίζουν στην θέση της κοινόχρηστης μνήμης που αντιστοιχεί στο προσωπικό αναγνωριστικό (ταυτότητα, ID number) του κάθε επεξεργαστή. Για παράδειγμα στο 'βήμα 1' ο επεξεργαστής 1 αποθηκεύει το άθροισμα που υπολόγισε στην πρώτη θέση του πίνακα, ενώ ο επεξεργαστής 2 στην δεύτερη θέση.

4.3.2 Πολυπλοκότητα Αλγ. Παράλληλης Άθροισης

Έστω ότι επιλύουμε ένα πρόβλημα μεγέθους (n).

Πλήθος Επεξεργαστών: $\mathrm{P}(n) = \rho = {}^{n}/_{2}$

Για τον υπολογισμό του παράλληλου χρόνου εκτέλεσης πρέπει να βρούμε το πλήθος των παράλληλων επαναλήψεων που εκτελούνται από τον αλγόριθμο παράλληλης άθροισης. Όπως προαναφέραμε στο υποκεφάλαιο 4.3.1 για ένα πρόβλημα αρχικού μεγέθους n, με το πέρας του πρώτου βήματος το μέγεθος των τιμών που πρέπει να αθροιστούν μειώνεται στο μισό $\left({}^{n}/_{2}\right)$. Γίνεται αυτή η επανάληψη έως ότου εναπομείνει μία τιμή. Το πλήθος των βημάτων (παράλληλων επαναλήψεων) που εκτελούνται είναι $\Theta(\log_2 n)$. Επίσης σε κάθε βήμα εκτελείται σταθερός αριθμός εντολών (δύο αναγνώσεις, μία άθροιση και μία εγγραφή). Συνεπώς ο παράλληλος χρόνος εκτέλεσης είναι το γινόμενο των δύο αυτών τιμών.

Παράλληλος Χρόνος Εκτέλεσης:

$$\begin{aligned}\mathrm{T}_{{}^{n}/_{2}}(n) &= (\textit{πλήθος επαναλήψεων}) \times (\textit{σταθερός αριθμός εντολών βήματος}) \\ &= \Theta(\log_2 n) \times \Theta(1) \\ &= \Theta(\log_2 n)\end{aligned}$$

Κόστος:

$$\begin{aligned}\mathrm{C}_{{}^{n}/_{2}}(n) &= \mathrm{T}_{{}^{n}/_{2}}(n) \times \mathrm{P}(n) \\ &= \Theta(\log_2 n) \times \left({}^{n}/_{2}\right) \\ &= \Theta(n \log_2 n)\end{aligned}$$

4.3.3 Διαφορετικές Υλοποιήσεις του Αλγ. Παράλληλης Άθροισης

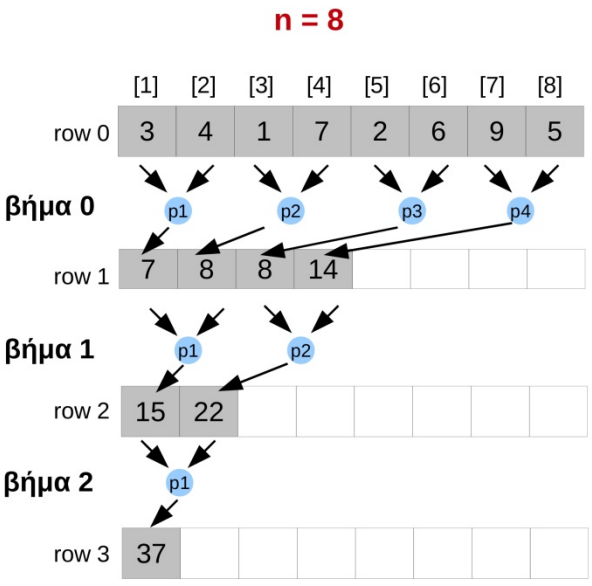
Πιο πριν αναφέραμε τους δύο άξονες γύρω από τους οποίους περιστρέφονται όλοι οι αλγόριθμοι. Ο πρώτος άξονας είναι η αναπαράσταση των δεδομένων στην μνήμη (memory mapping). Ο δεύτερος άξονας είναι το πώς γίνεται η επεξεργασία αυτών των δεδομένων (data processing). Για να μελετήσουμε την επίδραση του τρόπου οργάνωσης των δεδομένων στην μνήμη, στην επίδοση του αλγορίθμου· έχουμε ετοιμάσει διαφορετικές υλοποιήσεις με βάση τον αλγόριθμο που προαναφέραμε στο υποκεφάλαιο 4.3.1. Ο δεύτερος άξονας παραμένει σε γενικές γραμμές αναλλοίωτος (υπάρχουν

μερικές αλλαγές στις εντολές του αλγορίθμου, αλλά τα βασικά σημεία παραμένουν αναλλοίωτα). Στα επόμενα υποκεφάλαια θα παρουσιαστούν οι διαφορετικές υλοποιήσεις η κάθε μία ξεχωριστά, δίνοντας ιδιαίτερη σημασία στην διάταξη και διαχείριση των δεδομένων της κοινόχρηστης μνήμης.

4.3.3.1 Αλγ. Παράλληλης Άθροισης parSum-A

Σε αυτή την υλοποίηση ακολουθείται ακριβώς η ίδια διαδικασία στην ανάγνωση και γραφή όπως αυτή περιγράφεται στο υποκεφάλαιο 4.3.1, όμως με τη διαφορά ότι οι αναγνώσεις και οι εγγραφές γίνονται σε διαφορετική γραμμή. Ο λόγος για τον οποίο χρησιμοποιείτε διαφορετική γραμμή για την εγγραφή των αποτελεσμάτων, είναι εξαιτίας του ότι η πλατφόρμα XMT υλοποιεί το μοντέλο CRCW επομένως πρέπει να είμαστε προσεκτικοί κατά την αποθήκευση δεδομένων Για να κατανοήσουμε καλύτερα τις αλλαγές που υπάρχουν σε αυτή την υλοποίηση θα μελετήσουμε στην επόμενη παράγραφο ένα απλό παράδειγμα.

Έστω ότι το πρόβλημα άθροισης που επιθυμούμε να επιλύσουμε είναι μεγέθους $(n = 8)$. Όπως φαίνεται στο Σχήμα 4.2 αρχικά τα n στοιχεία βρίσκονται αποθηκευμένα στην πρώτη γραμμή του δυσδιάστατου πίνακα. Ο πίνακας έχει διαστάσεις $(n \times \log_2 n)$. Παρατηρήστε ότι κατά το πρώτο βήμα, 'βήμα 0', η ανάγνωση των τιμών γίνεται από την γραμμή 'row 0' του πίνακα. Οι εγγραφές γίνονται στην επόμενη γραμμή 'row 1'. Κατά το δεύτερο βήμα οι αναγνώσεις γίνονται από την γραμμή που στο προηγούμενο βήμα γίνονταν οι εγγραφές, δηλαδή την γραμμή 'row 1'. Ενώ οι εγγραφές γίνονται στην αμέσως επόμενη γραμμή από αυτήν που γίνονταν οι εγγραφές στο προηγούμενο βήμα.



Σχήμα 4.2 - Παράλληλη άθροιση parSum-A ($n = 8$)

Αξιοσημείωτο είναι το γεγονός ότι ο αλγόριθμος παραμένει σε αρκετά σημεία όμοιος με τον ψευδοκώδικα

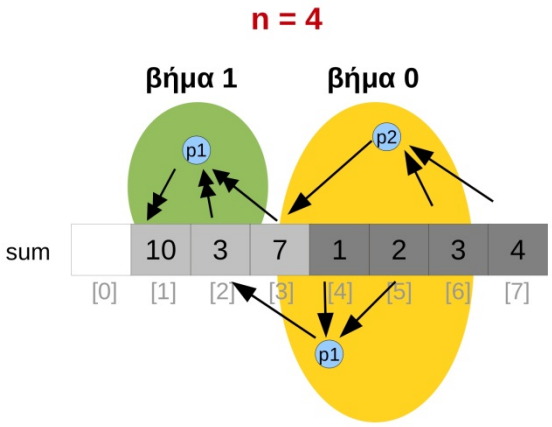
του αλγορίθμου παράλληλης άθροισης (Ψευδοκώδικας 4.2). Ολόκληρος ο κώδικας XMTC για την υλοποίηση του parSum-A παρατίθεται στο Παράρτημα Α.

4.3.3.2 Αλγ. Παράλληλης Άθροισης parSum-B

Ομοίως και σε αυτή την υλοποίηση οι αναγνώσεις και οι εγγραφές ακολουθούν την ίδια φιλοσοφία όπως αυτή περιγράφεται στο υποκεφάλαιο 4.3.1, όμως με τη διαφορά ότι τα δεδομένα αναπαριστούνται στην μνήμη ως δυαδικό δέντρο σε μονοδιάστατο πίνακα μεγέθους $(2n-1)$. Ακολουθεί παράδειγμα για να κατανοήσουμε καλύτερα την υλοποίηση αυτή.

Έστω ότι το πρόβλημα άθροισης που επιθυμούμε να επιλύσουμε είναι μεγέθους $(n=4)$. Όπως φαίνεται στο Σχήμα 4.3 αρχικά τα n στοιχεία βρίσκονται αποθηκευμένα στις τελευταίες n θέσεις/κελιά του μονοδιάστατου πίνακα. Ο πίνακας έχει μέγεθος $(2n-1)$. Παρατηρήστε ότι κατά το πρώτο βήμα, 'βήμα 0', η ανάγνωση των τιμών γίνεται από τα φύλλα του δυαδικού δέντρου, δηλαδή τα τελευταία n κελιά του πίνακα. Οι εγγραφές γίνονται στο κελί 'πατέρας' των φύλλων που αθροίζονται. Ακολούθως στο επόμενο βήμα τα κελιά στα οποία έχουν αποθηκευτεί τα αθροίσματα, θεωρούνται ως φύλλα και ακολουθείται η ίδια διαδικασία. Αυτό επαναλαμβάνεται μέχρι να αποθηκευτεί το συνολικό άθροισμα στην ρίζα του δέντρου.

Συγκρίνοντας τον ψευδοκώδικα της παράλληλης άθροισης (Ψευδοκώδικας 4.2) με τον κώδικα του parSum-B μπορούμε να παρατηρήσουμε ότι εξακολουθούν να υπάρχουν οι δύο αναγνώσεις, ο υπολογισμός και η εγγραφή. Απλώς αλλάζει η δομή των δεδομένων στην μνήμη. Ολόκληρος ο κώδικας XMTC για την υλοποίηση του parSum-B παρατίθεται στο Παράρτημα Α.

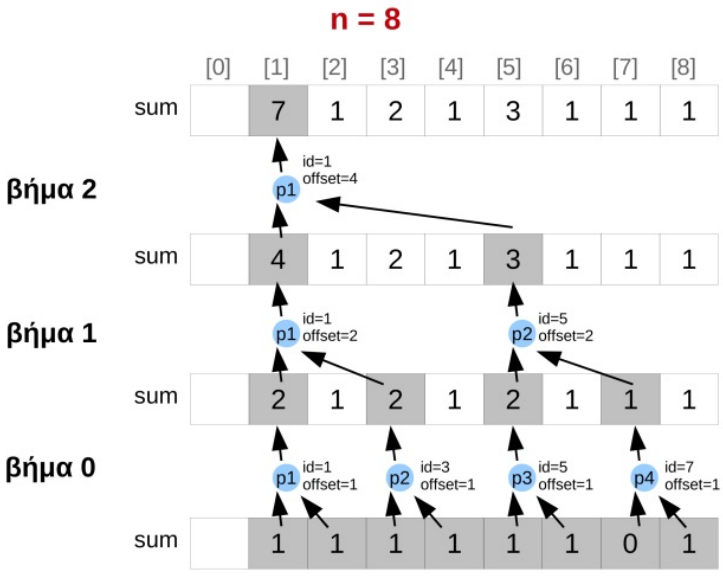


Σχήμα 4.3 – Παράλληλη άθροιση parSum-B ($n = 4$)

4.3.3.3 Αλγ. Παράλληλης Άθροισης parSum-C

Σε αυτή την υλοποίηση τα δεδομένα αναπαριστούνται στην μνήμη σε μονοδιάστατο πίνακα μεγέθους n. Ακολουθείται η ίδια φιλοσοφία ξανά όπως αυτή περιγράφεται στο υποκεφάλαιο 4.3.1. Η σημαντική διαφορά είναι ότι σε αυτή την υλοποίηση γίνεται έξυπνη χρήση της κοινόχρηστης μνήμης. Πιο συγκεκριμένα ο κάθε επεξεργαστής περιορίζεται εντός κάποιων ορίων, δηλαδή δικαιούται να διαβάσει τιμές και να γράψει δεδομένα μόνο σε συγκεκριμένες θέσεις. Ακολουθεί παράδειγμα το οποίο θα μας βοηθήσει να καταλάβουμε ακριβώς τι συμβαίνει.

Έστω ότι το πρόβλημα άθροισης που επιθυμούμε να επιλύσουμε είναι μεγέθους $(n = 8)$. Όπως φαίνεται στο Σχήμα 4.4 αρχικά τα n στοιχεία βρίσκονται αποθηκευμένα στον μονοδιάστατου πίνακα. Ο πίνακας έχει μέγεθος $(2n - 1)$. Παρατηρήστε ότι κατά το πρώτο βήμα, 'βήμα 0', ο κάθε επεξεργαστής διαβάζει, υπολογίζει και γράφει το άθροισμα σε ένα από τα κελιά που έχει χρησιμοποιήσει για διάβασμα. Συγκεκριμένα στο βήμα αυτό ο κάθε



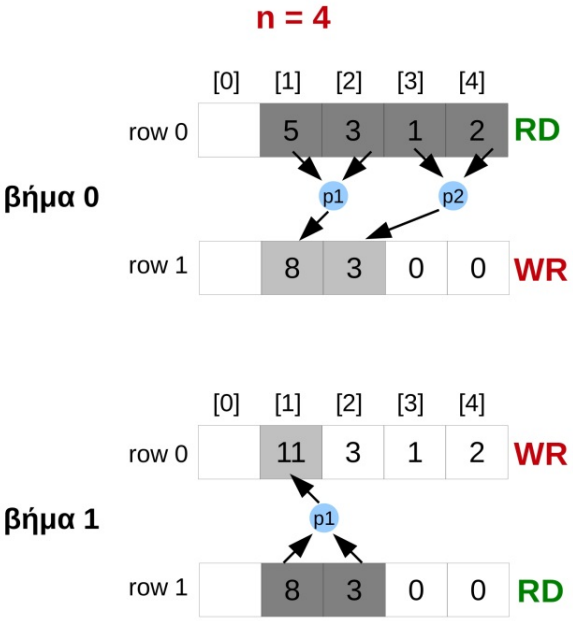
Σχήμα 4.4 – Παράλληλη άθροιση parSum-C ($n = 8$)

επεξεργαστής έχει πρόσβαση μόνο σε δύο κελιά. Το εύρος των κελιών που έχει πρόσβαση ο κάθε επεξεργαστής καθορίζεται από το πεδία ‘id’ του κάθε επεξεργαστή και την καθολική μεταβλητή ‘offset’. Παρομοίως με τις προηγούμενες υλοποιήσεις σε κάθε βήμα το πλήθος των αριθμών που πρέπει να αθροιστούν αλλά και των ενεργών επεξεργαστών υποδιπλασιάζεται έως ότου εκτελεστούν $\log_2 n$ βήματα.

Οι αναγνώσεις αριθμών, η άθροιση τους και η εγγραφή του αποτελέσματος στην κοινόχρηστη μνήμη εξακολουθεί να ακολουθεί την βασική δομή του παράλληλου αλγορίθμου άθροισης (Ψευδοκώδικας 4.2). Ολόκληρος ο κώδικας XMTC για την υλοποίηση του parSum-C παρατίθεται στο Παράρτημα Α.

4.3.3.4 Αλγ. Παράλληλης Άθροισης parSum-D

Σε αυτή την παραλλαγή του αλγορίθμου παράλληλης άθροισης τα δεδομένα αναπαριστούνται στην μνήμη σε δυσδιάστατο πίνακα διαστάσεων $(2 \times n)$. Εξακολουθεί να ισχύει η ίδια φιλοσοφία (δηλαδή χρησιμοποιείται ο ίδιος αριθμός επεξεργαστών, γίνεται ο ίδιος αριθμός αναγνώσεων και εγγραφών σε κάθε βήμα του αλγορίθμου). Αξίζει να σημειώσουμε ότι κανείς πρέπει να λάβει υπόψη του την υλοποίηση parSum-A όπως αυτή παρουσιάζεται στο υποκεφάλαιο 4.3.3.1. Η διαφορά σε αυτή την υλοποίηση είναι ότι δεν γίνεται χρήση ενός δυσδιάστατου πίνακα με διαστάσεις $(n \times \log_2 n)$ όπου σε κάθε βήμα γράφονται τα ενδιάμεσα αθροίσματα σε ξεχωριστή γραμμή (συγκεκριμένα εκτελούνται $\log_2 n$ βήματα). Εν μέρει γίνεται εναλλαγή στον χώρο που γράφονται τα ενδιάμεσα αποτελέσματα αλλά ακολουθεί το σκεπτικό ότι δεν πρέπει να μένουν γραμμές του πίνακα αναξιοποίητες. Εάν προσέξετε στην υλοποίηση του parSum-A σε κάθε βήμα του αξιοποιούνται μόνο δύο από τις $\log_2 n$ γραμμές του πίνακα. Η φιλοσοφία που ακολουθείται στην υλοποίηση του parSum-D είναι ότι θα υπάρχουν μόνο δύο γραμμές στον πίνακα και θα γίνεται εναλλαγή στον ρόλο της κάθε γραμμής



Σχήμα 4.5 – Παράλληλη άθροιση parSum-C ($n = 8$)

σε κάθε βήμα. Με τον όρο ‘ρόλος της κάθε γραμμής’ εννοούμε τη χρήση της κάθε γραμμής, δηλαδή αν η συγκεκριμένη γραμμή χρησιμοποιείται για ανάγνωση ή για αποθήκευση αποτελεσμάτων. Στην επόμενη παράγραφο θα μελετήσουμε ένα απλό σενάριο για τον parSum-D προκειμένου να κατανοήσουμε καλύτερα τον τρόπο λειτουργίας.

Έστω ότι το πρόβλημα άθροισης που επιθυμούμε να επιλύσουμε είναι μεγέθους $(n = 4)$. Όπως φαίνεται στο Σχήμα 4.5 αρχικά τα n στοιχεία βρίσκονται αποθηκευμένα στον δυσδιάστατο πίνακα $(2 \times n)$. Πιο συγκεκριμένα βρίσκονται αποθηκευμένα στην πρώτη γραμμή του πίνακα δηλαδή την ‘row 0’. Παρατηρήστε ότι κατά το πρώτο βήμα, ‘βήμα 0’, ο κάθε επεξεργαστής διαβάζει από την ‘row 0’ τους δύο αριθμούς που του αναλογούν, υπολογίζει το άθροισμά τους και το αποθηκεύει στο αντίστοιχο κελί της άλλης γραμμής, δηλαδή της ‘row 1’. Στο Σχήμα 4.5 εμφανίζεται δεξιά από κάθε γραμμή ο ρόλος της. Κατά το πρώτο βήμα η γραμμή ‘row 0’ χρησιμοποιείται για ανάγνωση (RD - read) ενώ η γραμμή ‘row 1’ χρησιμοποιείται για αποθήκευση των αποτελεσμάτων (WR - write). Με την ολοκλήρωση του πρώτου βήματος γίνεται εναλλαγή των ρόλων των γραμμών, δηλαδή η ‘row 0’ χρησιμοποιείται για αποθήκευση δεδομένων (WR - write) ενώ η ‘row 1’ χρησιμοποιείται για ανάγνωση (RD - read) των ενδιάμεσων αποτελεσμάτων. Η εναλλαγή των ρόλων των γραμμών του πίνακα είναι ιδιαίτερα σημαντική μιας και αποτελεί το συνεκτικό στοιχείο μεταξύ των $\log_2 n$ βημάτων που εκτελούνται. Όπως και στις προηγούμενες υλοποιήσεις σε κάθε βήμα το πλήθος των αριθμών που πρέπει να αθροιστούν αλλά και των ενεργών επεξεργαστών υποδιπλασιάζεται έως ότου εκτελεστούν $\log_2 n$ βήματα.

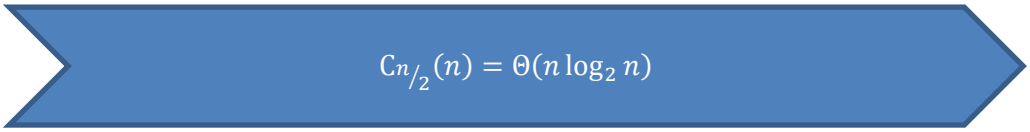
Οι αναγνώσεις αριθμών, η άθροιση τους και η εγγραφή του αποτελέσματος στην κοινόχρηστη μνήμη εξακολουθεί να ακολουθεί την βασική δομή του παράλληλου αλγορίθμου άθροισης (Ψευδοκώδικας 4.2). Ολόκληρος ο κώδικας XMTC για την υλοποίηση του parSum-D παρατίθεται στο Παράρτημα Α.

4.4 Βέλτιστος Αλγ. Παράλληλης Άθροισης

Παρόλο που αλγόριθμος παράλληλης άθροισης που προαναφέραμε επιλύει το πρόβλημα της άθροισης σε χρόνο $T_{n/2}(n) = \Theta(\log_2 n)$ δεν είναι κόστους-βέλτιστος αλγόριθμος (βλέπε υποκεφάλαιο 2.2). Για να θεωρείται ένας παράλληλος αλγόριθμος

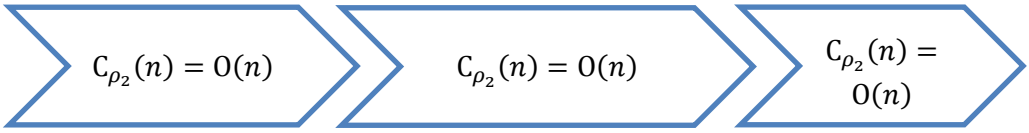
κόστους-βέλτιστος πρέπει να ισχύει $C_\rho(n) = O\left(\tau_\rho(n)\right)$, όπου $\tau_\rho(n)$ ο χρόνος εκτέλεσης της καλύτερης σειριακής λύσης. Η καλύτερη σειριακή λύσει έχει χρόνο $\tau_1(n) = \Theta(n)$. Ο αλγόριθμος παράλληλης άθροισης έχει κόστος $C_{n/2}(n) = \Theta(n \log_2 n)$. Άρα, $C_\rho(n) \notin O(n)$.

Πριν προχωρήσουμε ας μελετήσουμε το Σχήμα 4.6 για να κατανοήσουμε καλύτερα το σκεπτικό πίσω από την βέλτιστη λύση. Στο Σχήμα 4.6(α) παρουσιάζεται η μονολιθική επίλυση του προβλήματος άθροισης με την εφαρμογή του αλγορίθμου παράλληλης άθροισης. Αυτό που επιθυμούμε να επιτύχουμε είναι η μείωση του κόστους του αλγορίθμου στο σύνολο του. Δηλαδή εάν είναι δυνατόν να επιλύσουμε το πρόβλημα άθροισης όχι μονολιθικά αλλά λύνοντας το σε περισσότερα από ένα στάδια (δες Σχήμα 4.6(α) και (β)), με συνολικό κόστος $O(n)$ τότε θα έχουμε επιτύχει τον στόχο μας για κόστους-βέλτιστη λύση. Σημαντικό σημείο είναι ότι πρέπει στην προτεινόμενη βέλτιστη λύση να συμπεριλαμβάνεται και ο αλγόριθμος παράλληλης άθροισης (δηλαδή ένα από τα στάδια πρέπει να είναι ο αλγόριθμος παράλληλης άθροισης). Εξαιτίας του ότι με το πέρας του αλγορίθμου παράλληλης άθροισης παραμένει μόνο μία τελική τιμή, είναι κατανοητό ότι ο αλγόριθμος παράλληλης άθροισης θα είναι το τελευταίο στάδιο της λύσης. Αναφερόμενος στην λύση που θα μελετήσουμε αυτή περιγράφεται αφαιρετικά από το Σχήμα 4.6(β) γιατί και η βέλτιστη λύση που θα αναλύσουμε περιλαμβάνει δύο στάδια.



(α) Αλγόριθμος Παράλληλης Άθροισης

(β) Πιθανή βέλτιστη λύση i



(γ) Πιθανή βέλτιστη λύση ii

Σχήμα 4.6 - Αφαιρετική όψη σταδίων επίλυσης προβλήματος άθροισης

Επειδή πρέπει στην βελτιωμένη έκδοση του αλγορίθμου να συμπεριλαμβάνεται το στάδιο της παράλληλης άθροισης πρέπει να φράξουμε το κόστος της σε $O(n)$. Φράζοντας το κόστος της παράλληλης άθροισης θα είμαστε σε θέση να αναγνωρίσουμε τα εναπομείναντα κομμάτια δουλειάς που πρέπει να ολοκληρώσουμε με κόστος $O(n)$.

Γνωρίζουμε ότι $C_\rho(n) = T_\rho(n) \times \rho$

Θέλουμε το κόστος μας να είναι $C_\rho(n) = O(n)$

Επίσης γνωρίζουμε ότι ο παράλληλος χρόνος ολοκλήρωσης για τον αλγόριθμο παράλληλης άθροισης δύνεται από την σχέση:

$$T_{n/2}(n) = \Theta(\log_2 n)$$

Ο λόγος για τον οποίο χρησιμοποιείται η τιμή αυτή είναι για να διασφαλίσουμε ότι το στάδιο της παράλληλης άθροισης θα έχει κόστος $O(n)$.

Άρα,

$$\Theta(\log_2 n) \times M = O(n)$$

$$\Rightarrow \mathrm{M} = \frac{n}{\log_2 n}$$

Το πλήθος των επεξεργαστών M που προκύπτει αποτελεί τον μέγιστο δυνατό αριθμό επεξεργαστών που μπορούμε να χρησιμοποιήσουμε στην λύση μας. Προσπαθούμε να ολοκληρώσουμε όσο το δυνατόν μεγαλύτερο μέρος της δουλειάς γίνεται από το πρώτο στάδιο (το στάδιο αυτό ονομάζεται 'σειριακό' ή 'στάδιο μείωσης εργασίας').

Συγκεκριμένα για το πρώτο στάδιο έχουμε στην διάθεση μας M επεξεργαστές τους οποίους επιθυμούμε να χρησιμοποιήσουμε για να αθροίσουμε μέρος των αριθμών από το αρχικό σύνολο των n αριθμών. Σε αυτό το σημείο πρέπει να αποφασίσουμε πώς θα αξιοποιήσουμε τους επεξεργαστές για να πετύχουμε ολοκλήρωση του μέγιστου δυνατού όγκου εργασίας. Το όριο που πρέπει να λάβουμε υπόψη είναι το $\mathrm{O}(n)$. Ταυτοχρόνως και οι M επεξεργαστές εκτελούν σειριακή άθροιση $\log_2 n$ αριθμών ο κάθε ένας ξεχωριστά και αποθηκεύουν το τελικό αποτέλεσμα στις αριστερές θέσεις του πίνακα (εδώ μπορεί να χρειαστούν κάποια επιπλέον βήματα για patching εξαιτίας του ότι $(n - n/\log_2 n \neq 0)$, λόγω του ότι ο αριθμός των βημάτων είναι σταθερού αριθμού και συγκεκριμένα μικρότερος από $n/\log_2 n$ μπορούμε να τα αγνοήσουμε). Η πολυπλοκότητα του πρώτου σταδίου φαίνεται πιο κάτω:

<u>Πλήθος Επεξεργαστών:</u> $\mathrm{P}(n) = \rho = \frac{n}{\log_2 n}$

<u>Παράλληλος Χρόνος Εκτέλεσης:</u> $\mathrm{T}_\rho(n) = \Theta(\log_2 n)$

<u>Κόστος:</u>
$$\begin{aligned} \mathrm{C}_\rho(n) &= \mathrm{T}_\rho(n) \times \mathrm{P}(n) \\ &= \log_2 n \times \frac{n}{\log_2 n} \\ &= \mathrm{O}(n) \end{aligned}$$

Με το πέρας στου πρώτου σταδίου εναπομένουν $\mathrm{M} = \frac{n}{\log_2 n}$ ενδιάμεσα αποτελέσματα τα οποία πρέπει να αθροιστούν μαζί. Παρατηρούμε ότι για την άθροιση των αριθμών με την χρήση του αλγορίθμου παράλληλης άθροισης χρειάζονται το πολύ $\mathrm{M}/2$ επεξεργαστές. Αυτό το τελευταίο βήμα ονομάζεται 'βήμα παράλληλης άθροισης'.

Υπολογίζοντας την πολυπλοκότητα του αλγορίθμου παράλληλης άθροισης με τα νέα δεδομένα προκύπτουν τα εξής:

Πλήθος Επεξεργαστών: $\mathrm{P}\left(\frac{n}{\log_2 n}\right) = \rho = \frac{n}{2\log_2 n}$

Παράλληλος Χρόνος Εκτέλεσης: $\mathrm{T}_\rho\left(\frac{n}{\log_2 n}\right) = \Theta\left(\log_2\left(\frac{n}{\log_2 n}\right)\right)$

Κόστος:
$$\begin{aligned}\mathrm{C}_\rho\left(\frac{n}{\log_2 n}\right) &= \mathrm{T}_\rho\left(\frac{n}{\log_2 n}\right) \times \rho \\ &= \log_2\left(\frac{n}{\log_2 n}\right) \times \frac{n}{2\log_2 n} \\ &= \mathrm{O}(n)\end{aligned}$$

4.4.1 Πολυπλοκότητα Βέλτιστου Αλγ. Παράλληλης Άθροισης

Από τα προαναφερθέν στοιχεία του υποκεφαλαίου 4.4 η πολυπλοκότητα του βέλτιστου αλγορίθμου παράλληλης άθροισης είναι η εξής:

Πλήθος Επεξεργαστών: $\mathrm{P}\left(\frac{n}{\log_2 n}\right) = \rho = \frac{n}{\log_2 n}$

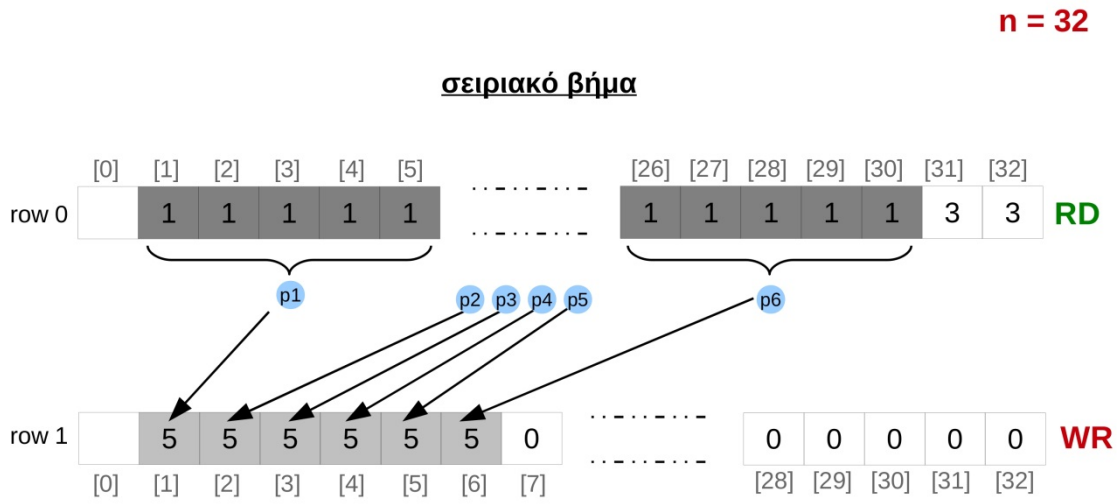
Παράλληλος Χρόνος Εκτέλεσης: $\mathrm{T}_\rho(n) = \Theta(\log_2 n)$

Κόστος: $\mathrm{C}_\rho(n) = \mathrm{O}(n)$

Ο χρόνος εκτέλεσης του παράλληλου βήματος είναι $\log_2\left(\frac{n}{\log_2 n}\right)$, ενώ του σειριακού βήματος $\Theta(\log_2 n)$. Ως αποτέλεσμα η τάξη του χρόνου του σειριακού βήματος υπερνικά το παράλληλο βήμα. Όπως φαίνεται ο βέλτιστος αλγόριθμος είναι και κόστους-βέλτιστος αφού επιλύει το πρόβλημα μεγέθους n με κόστος $\mathrm{O}(n)$ το οποίο είναι της τάξης όμικρον του χρόνου της καλύτερης σειριακής εκτέλεσης.

4.4.2 Αλγ. Παράλληλης Άθροισης parSum-D.optimized

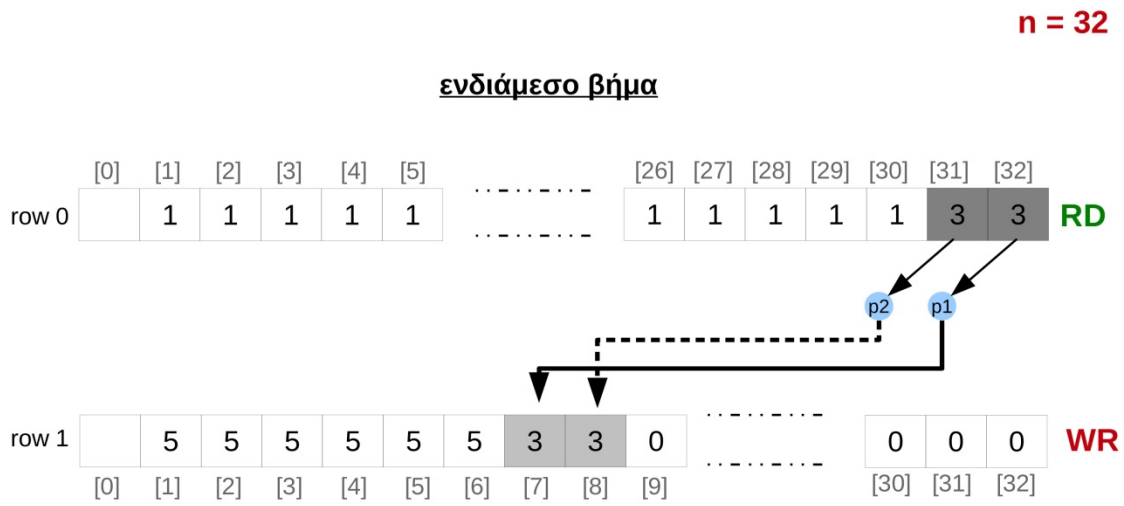
Σε αυτή την υλοποίηση χρησιμοποιήθηκε ως βάση ο ήδη υλοποιημένος αλγόριθμος παράλληλης άθροισης parSum-D προκειμένου να υλοποιηθεί ο βέλτιστος αλγόριθμος παράλληλης άθροισης. Στην περιγραφή που ακολουθεί περιγράφεται σύντομα ο τρόπος λειτουργίας του αλγορίθμου.



Σχήμα 4.7 – Βέλτιστη παράλληλη άθροιση – σειριακό βήμα ($n = 32$)

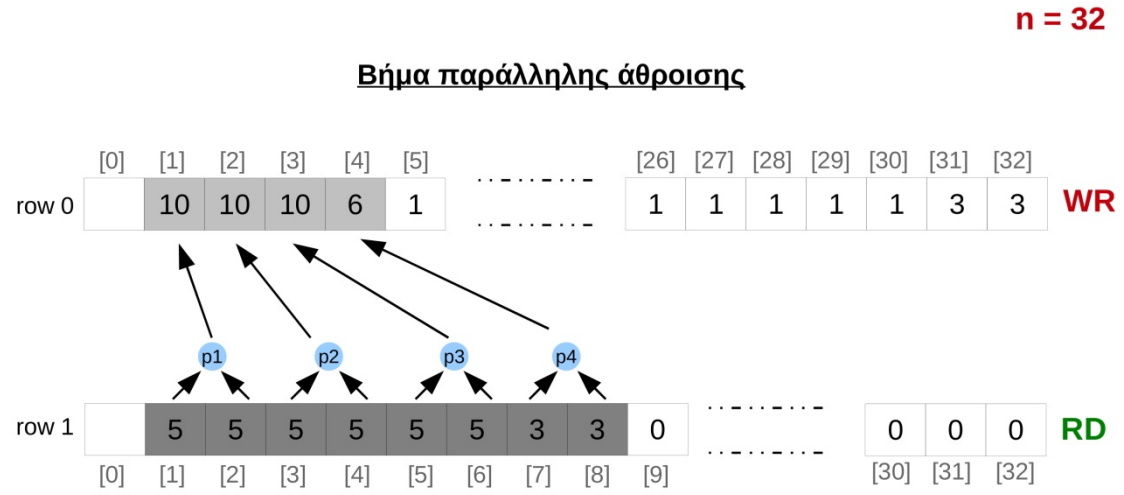
Έστω ότι το πρόβλημα άθροισης που επιθυμούμε να επιλύσουμε είναι μεγέθους $(n = 32)$. Όπως φαίνεται στο Σχήμα 4.7 αρχικά τα n στοιχεία βρίσκονται αποθηκευμένα στην πρώτη γραμμή του δυσδιάστατου πίνακα. Ο πίνακας έχει διαστάσεις $(2 \times n)$. Παρατηρήστε ότι κατά το σειριακό βήμα, υπάρχουν $(n/\log_2 n)$ ενεργοί επεξεργαστές οι οποίοι αθροίζουν από $\log_2 n$ στοιχεία ο κάθε ένας ταυτόχρονα. Τα επί μέρους αθροίσματα αποθηκεύονται στις πρώτες $\log_2 n$ θέσεις του πίνακα βάσει του προσωπικού αναγνωριστικού του κάθε επεξεργαστή. Οι εγγραφές γίνονται στην επόμενη γραμμή 'row 1'. Με την ολοκλήρωση των εγγραφών τελειώνει το σειριακό βήμα.

Μετά το σειριακό βήμα ακολουθεί το ενδιάμεσο βήμα το οποίο έχει ως σκοπό να τοποθετήσει τα στοιχεία που δεν έχουν αθροιστεί από κάποιο επεξεργαστή (κατά την εκτέλεση του σειριακού βήματος) στα κελιά μνήμης που βρίσκονται δεξιά των πρώτων $\log_2 n$ κελιών της γραμμής 'row 1'. Στο Σχήμα 4.8 παρουσιάζεται γραφικά το τι συμβαίνει κατά το βήμα αυτό.



Σχήμα 4.8 – Βέλτιστη παράλληλη άθροιση – ενδιάμεσο βήμα – patching ($n = 32$)

Με την ολοκλήρωση του ενδιάμεσου βήματος γίνεται αντιστροφή των ρόλων των γραμμών, δηλαδή η γραμμή ‘row 1’ θα χρησιμοποιείται για ανάγνωση (RD-read) ενώ η γραμμή ‘row 0’ θα χρησιμοποιείται για εγγραφές (WR-write). Ακολούθως εφαρμόζεται ο αλγόριθμος παράλληλης άθροισης (βλέπε Σχήμα 4.9) ακολουθώντας την ίδια μεθοδολογία με την υλοποίηση parSum-D. Δηλαδή σε με την ολοκλήρωση κάθε



Σχήμα 4.9 – Βέλτιστη παράλληλη άθροιση – βήμα παράλληλη άθροισης ($n = 32$)

βήματος γίνεται εναλλαγή των ρόλων των γραμμών του πίνακα και το πλήθος των επεξεργαστών που παραμένουν ενεργοί υποδιπλασιάζεται. Υποδιπλασιασμός συμβαίνει και στο πλήθος των ενδιάμεσων αθροισμάτων. Η μόνη διαφορά που υπάρχει είναι ότι λαμβάνεται υπόψη η περίπτωση που με την ολοκλήρωση του ενδιάμεσου βήματος προκύπτει μονός αριθμός τιμών για άθροιση. Τότε ο πρώτος επεξεργαστής μεριμνά

ειδικά για αυτή την περίπτωση και χρεώνεται με μία επιπλέον πράξη (αυτό λαμβάνει χώρα μόνο μία φορά).

Ολόκληρος ο κώδικας XMTC για την υλοποίηση του parSum-D.optimized παρατίθεται στο Παράρτημα Α.

4.5 Πειραματική Αξιολόγηση Υλοποιήσεων και Μετρικές

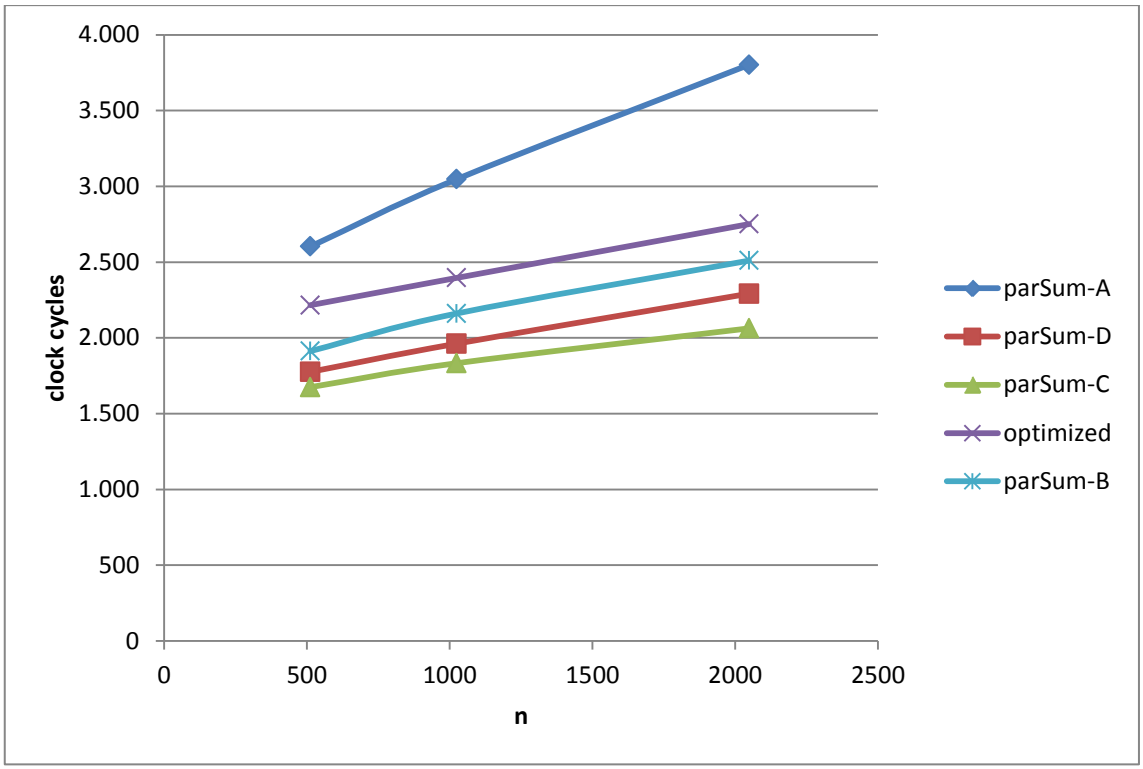
Οι αλγόριθμοι που αναφέραμε στο υποκεφάλαια 4.3.3 και 4.4 υλοποιήθηκαν και προσομοιώθηκαν στην πλατφόρμα XMT. Να τονίσουμε ότι και οι πέντε διαφορετικές υλοποιήσεις προσομοιώθηκαν για προβλήματα μεγέθους n=512, 1024 και 2048. Επίσης μια προσομοίωση είναι αρκετή για την λήψη μετρήσεων, αφού πολλαπλές διαδοχικές προσομοιώσεις για το ίδιο μέγεθος προβλήματος δίδουν το ίδιο αποτέλεσμα.

Στον πιο κάτω πίνακα (Πίνακας 4.1) παρουσιάζονται οι μετρήσεις που καταγράφηκαν για τους κύκλους ολοκλήρωσης του κάθε αλγορίθμου ξεχωριστά.

n	parSum-A	parSum-D	parSum-C	cost.optim	parSum-B
512	2.603	1.776	1.675	2.216	1.913
1024	3.047	1.962	1.833	2.396	2.161
2048	3.802	2.292	2.063	2.751	2.510

Πίνακας 4.1 – Clock cycles ανά αλγόριθμο

Λαμβάνοντας υπόψη τις μετρήσεις (Πίνακας 4.1) και το Γράφημα 4.1 παρατηρούμε ότι υπάρχουν διαφορές ως προς τον συνολικό χρόνο ολοκλήρωσης των αλγορίθμων. Συγκεκριμένα τους περισσότερους κύκλους για ολοκλήρωση έχει η υλοποίηση parSum-A η οποία έχει την μεγαλύτερη απαίτηση σε μνήμη. Συγκεκριμένα ο parSum-A απαιτεί χώρο $(n \times \log_2 n)$. Αντιθέτως η υλοποίηση parSum-C η οποία έχει απαιτεί μόνο n θέσεις μνήμης σημειώνει λιγότερους κύκλους. Στο ενδιάμεσο των προαναφερθέντων υλοποιήσεων βρίσκονται οι parSumD $(2n)$, parSum-B $(2n-1)$ και parSum.optimized $(2n)$. Παρατηρείστε ότι αυτές οι υλοποιήσεις έχουν τις ίδιες απαιτήσεις σε μνήμη ενώ συγκρίνοντας τους κύκλους τους με τον ελάχιστο και τον μέγιστο αριθμό κύκλων, αυτές κυμαίνονται πιο κοντά προς τους ελάχιστους. Το ότι η υλοποίηση parSum.optimized βρίσκεται πιο πάνω από τις parSum-B, parSum-C και parSum-D είναι εξαιτίας του σειριακού βήματος το οποίο χρεώνει την υλοποίηση με κύκλους.



Γράφημα 4.1 – Γραφική αναπαράσταση των μετρήσεων clock cycles (Πίνακας 4.1)

Ο Πίνακας 4.2 παρουσιάζει τον μέγιστο αριθμό επεξεργαστών που χρησιμοποιεί η κάθε υλοποίηση. Εξαιτίας του περιορισμού στο πλήθος των thread που θέτει η πλατφόρμα XMT περιοριζόμαστε στο μέγεθος των προβλημάτων που μπορούμε να προσομοιώσουμε. Αξίζει να τονίσουμε το γεγονός ότι εξαιτίας της ιδιότητας της υλοποίησης parSum-D.optimized να είναι και κόστους-βέλτιστη είναι δυνατή η προσομοίωση προβλημάτων μεγαλύτερου μεγέθους. Συγκεκριμένα οι υπόλοιπες υλοποιήσεις πέραν του parSum-D.optimized, μπορούν να λύσουν προβλήματα μεγέθους έως $(n = 2048)$. Ενώ η υλοποίηση parSum-D.optimized δύναται να επιλύσει προβλήματα μεγέθους έως $(n \approx 8192)$.

n	parSum-A	parSum-D	parSum-C	cost.optim	parSum-B
512	256	256	256	56	256
1024	512	512	512	102	512
2048	1.024	1.024	1.024	186	1.024

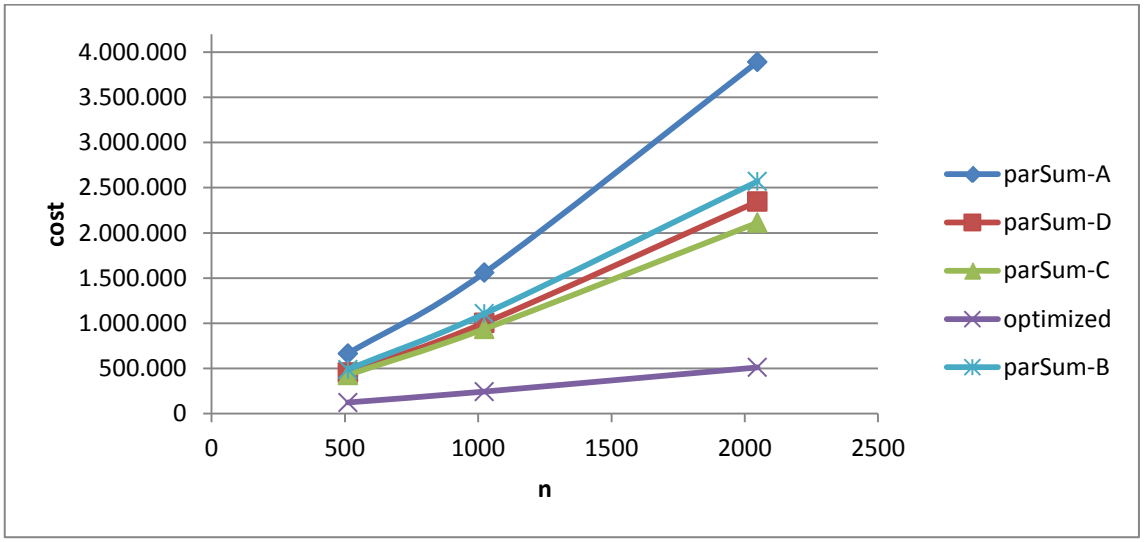
Πίνακας 4.2 – Πλήθος επεξεργαστών (ρ) ανά αλγρόριθμο

Πέραν των μετρήσεων για clock cycles και πλήθη επεξεργαστών έχουν ληφθεί μετρήσεις για το κόστος του κάθε αλγορίθμου ανά μέγεθος προβλήματος ξεχωριστά. Στον Πίνακας 4.3 φαίνονται οι μετρήσεις που λήφθηκαν ενώ στο Γράφημα 4.2 παρουσιάζεται η γραφική αναπαράσταση τους. Το μεγαλύτερο κόστος το έχει η

n	parSum-A	parSum-D	parSum-C	cost.optim	parSum-B
512	666.368	454.656	428.800	124.096	489.728
1024	1.560.064	1.004.544	938.496	244.392	1.106.432
2048	3.893.248	2.347.008	2.112.512	511.686	2.570.240

Πίνακας 4.3 – Κόστος ανά αλγόριθμο

υλοποίηση parSum-A κυρίως λόγο του αυξημένου αριθμού clock cycle, εξαιτίας της μεγάλης ανάγκης σε μνήμη. Τα χαμηλότερα κόστη τα καταγράφει η υλοποίηση parSum-D.optimized αφού εξαιτίας του ότι είναι κόστους-βέλτιστος ως προς το χρόνο του καλύτερου σειριακού αλγορίθμου δίκαια τα καταφέρνει. Στο ενδιάμεσο των αλγορίθμων parSum-A και parSum-D.optimized κατατάσσονται οι υπόλοιπες υλοποιήσεις. Αξίζει να τονίσουμε το γεγονός ότι και οι τρείς αυτές υλοποιήσεις έχουν τις ίδιες απαιτήσεις σε μνήμη εκτός της υλοποίηση parSum-C της οποίας η απαιτήση σε μνήμη είναι n. Επίσης σημειώνει χαμηλότερο κόστος από τις υλοποιήσεις parSum-B και parSum-D αλλά η διαφορά θεωρείται μικρή εάν λάβουμε υπόψη όλες τις υλοποιήσεις μαζί.



Γράφημα 4.2 – Γραφική αναπαράσταση των μετρήσεων κόστους (Πίνακας 4.2)

Από τις μετρήσεις που λήφθηκαν κάναμε ένα υπολογισμό για τον καθορισμό της σταθεράς για κάθε υλοποίηση ξεχωριστά. Στον πιο κάτω πίνακα (Πίνακας 4.4) παρουσιάζονται οι τιμές για τις σταθερές. Να τονίσουμε δεν καταφέραμε να υπολογίσουμε ακριβώς την τιμή της σταθεράς για κάθε υλοποίηση αλλά πετύχαμε την ανάδειξη κάποιων τιμών γύρω από τις οποίες κυμαίνεται.

Algorithm	n	cost	f	Constant=cost / f	
parSum-B	512	489.728	4.608	106,27777777778	f = nlgn
parSum-B	1.024	1.106.432	10.240	108,05000000000	
parSum-B	2.048	2.570.240	22.528	114,09090909091	
cost.optim	512	124.096	512	242,37500000000	f = n
cost.optim	1.024	244.392	1.024	238,66406250000	
cost.optim	2.048	511.686	2.048	249,84667968750	
cost.optim	4.096	1.061.874	4.096	259,24658203125	
cost.optim	8.192	2.304.540	8.192	281,31591796875	
parSum-C	512	428.800	4.608	93,05555555556	f = nlgn
parSum-C	1.024	938.496	10.240	91,65000000000	
parSum-C	2.048	2.112.512	22.528	93,77272727273	
parSum-D	512	454.656	4.608	98,66666666667	f = nlgn
parSum-D	1.024	1.004.544	10.240	98,10000000000	
parSum-D	2.048	2.347.008	22.528	104,18181818182	
parSum-A	512	666.368	4.608	144,61111111111	f = nlgn
parSum-A	1.024	1.560.064	10.240	152,35000000000	
parSum-A	2.048	3.893.248	22.528	172,81818181818	

Πίνακας 4.4 – Υπολογισμός σταθερών

4.6 Συμπεράσματα

Η πειραματική αξιολόγηση διαφορετικών υλοποιήσεων του αλγορίθμου παράλληλης άθροισης στην πλατφόρμα XMT μας βοήθησε στην εξαγωγή συμπερασμάτων και διαπιστώσεων. Όπως είχαμε αναφέρει στους στόχους της Ατομικής Διπλωματικής Εργασίας (δες υποκεφάλαιο 1.2) θέλαμε να εξακριβώσουμε το βαθμό που επηρεάζει η διάταξη των δεδομένων στην μνήμη τις εκτελέσεις των αλγορίθμων στην πλατφόρμα XMT. Όπως φάνηκε από τις προσομοιώσεις των αλγορίθμων παράλληλης άθροισης (parSum-A, parSum-B, parSum-C, parSum-D και parSum-D.optimized) οι αυξημένες ανάγκες σε μνήμη αυξάνουν σημαντικά το χρόνο ολοκλήρωσης του αλγορίθμου και κατ' επέκταση το κόστος τους. Επίσης η έξυπνη διαχείριση της μνήμης (parSum-C) οδηγεί σε μειωμένο χρόνο ολοκλήρωσης και συνολικό κόστος. Ακόμη η υλοποίηση ενός βέλτιστου αλγορίθμου (parSum-D.optimized) ο οποίος εάν είναι και κόστους-βέλτιστος τότε επιτυγχάνεται η καλύτερη δυνατή αξιοποίηση των δυνατοτήτων της πλατφόρμας XMT.

Κεφάλαιο 5

Αλγόριθμος W

5.1 Το Πρόβλημα Write-All στο F-PRAM

Το πρόβλημα Write-All [4] ορίζεται ως εξής: Δεδομένου μιας λίστας n στοιχείων, όπου αρχικά όλα τα στοιχεία έχουν την τιμή 0, ζητείται να μετατραπούν οι τιμές των στοιχείων σε 1 χρησιμοποιώντας ρ επεξεργαστές στο μοντέλο F-PRAM. Ο μέγιστος αριθμός επεξεργαστών που μπορούν να καταρρεύσουν είναι f όπου $f < \rho$.

5.2 Περιγραφή Αλγορίθμου W

Ο Αλγόριθμος W [1] είναι μέχρι σήμερα ο καλύτερος αλγόριθμος για την επίλυση του προβλήματος Write-All. Ο αλγόριθμος εκτελεί ένα βρόγχο που αποτελείται από τέσσερις φάσεις και εκτελείται συνεχώς έως ότου λυθεί το πρόβλημα. Ο Ψευδοκώδικας 5.1 παρουσιάζει τη δομή του αλγορίθμου W. Τονίζεται ότι ο παρών αλγόριθμος έχει ανοχή σφαλμάτων και οι επεξεργαστές που παρουσιάζουν σφάλμα καταρρέουν μόνιμα (δεν επανέρχονται στους υπολογισμούς). Στις επόμενες παραγράφους ακολουθούν οι περιγραφές και η ανάλυση των φάσεων του αλγορίθμου. Επίσης να αναφέρουμε ότι η υλοποίηση του αλγορίθμου W (κώδικας) βρίσκεται στο Παράρτημα Β.

Φάση W1: γνωστή και ως η φάση καταμέτρησης, έχει ως κύριο στόχο την καταμέτρηση των ενεργών επεξεργαστών. Χρησιμοποιώντας ένα βοηθητικό διάνυσμα δ_1, το οποίο

αναπαριστά ένα νοητό δυαδικό δέντρο – το διάνυσμα είναι αποθηκευμένο στην μνήμη ως σωρός, οι επεξεργαστές ξεκινούν από τα φύλλα του δέντρου και εφαρμόζοντας μια εκδοχή του αλγορίθμου παράλληλης άθροισης (για το μοντέλο PRAM) αθροίζουν το πλήθος των επεξεργαστών που δεν έχουν καταρρεύσει. Αξίζει να σημειωθεί ότι πριν οι επεξεργαστές ξεκινήσουν να τρέχουν τον αλγόριθμο παράλληλης άθροισης κατανέμονται στα φύλλα βάση του προσωπικού κωδικού τους. Δηλαδή ο επεξεργαστής με κωδικό 1 βρίσκεται στο πρώτο φύλλο, ο επεξεργαστής με κωδικό 2 στο δεύτερο φύλλο και ούτω καθ' εξής. Με την ολοκλήρωση της παράλληλης άθροισης στην ρίζα του νοητού δυαδικού δέντρου (στην πρώτη θέση του διανύσματος δ_1) βρίσκεται αποθηκευμένο το πλήθος των ενεργών επεξεργαστών. Αυτός ο αριθμός αποτελεί μια υπερεκτίμηση για το πλήθος των ενεργών επεξεργαστών. Ο λόγος για τον οποίο συμβαίνει αυτό είναι γιατί μπορεί κάποιος/κάποιοι επεξεργαστές να συνεισφέρουν κατά την άθροιση αλλά αμέσως μετά να καταρρεύσουν με αποτέλεσμα το συνολικό άθροισμα να μην ανταποκρίνεται στην πραγματικότητα.

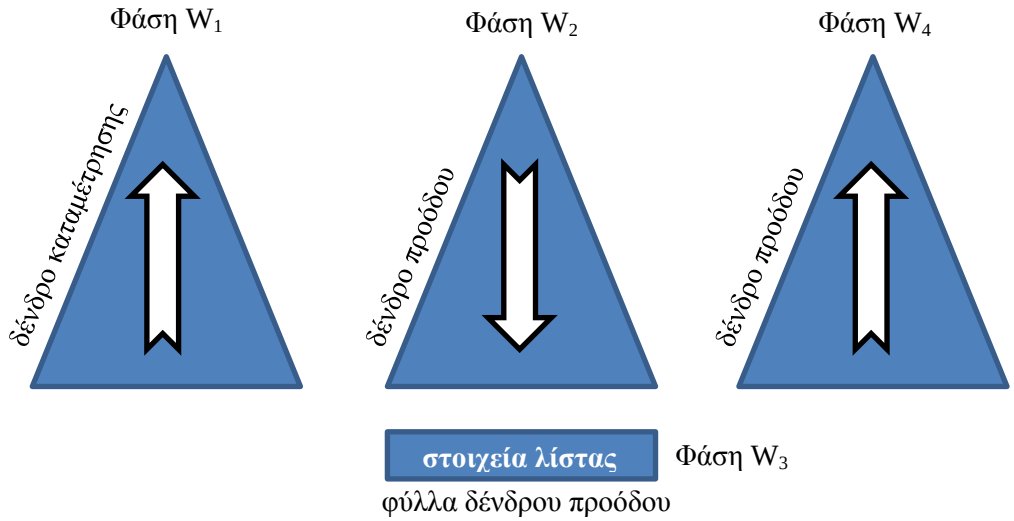
```
PRAM Pseudo-Code

LN
01    Processors i = 1 to n do in parallel:
02       Phase W3
03       Phase W4
04       While (done[1]!=n) do
05          Phase W1
06          Phase W2
07          Phase W3
08          Phase W4
09       End while
10    End do
```

Ψευδοκώδικας 5.1 – Αλγόριθμος W [1]

Φάση W2: αυτή η φάση ονομάζεται και φάση ανάθεσης εργασίας. Ο λόγος για το οποίο ονομάζεται έτσι είναι διότι με το πέρας της φάσης αυτής όσοι επεξεργαστές είναι ενεργοί ανατίθενται στα φύλλα που υπάρχει δουλειά για να ολοκληρωθεί. Όπως και στην Φάση W1 έτσι και εδώ γίνεται χρήση ενός διανύσματος δ_2 το οποίο αναπαριστά ένα δυαδικό δέντρο (αναπαρίσταται στην μνήμη ως σωρός), και επιπλέον συνδυάζεται γνώση από την Φάση W1 (διάνυσμα με πληροφορίες για τους ενεργούς επεξεργαστές) αλλά και τα μηδενικά στο δέντρο προόδου. Με την ολοκλήρωση της φάσης αυτής όσοι επεξεργαστές είναι ενεργοί είναι κατανεμημένοι σε φύλλα στα οποία πρέπει να γίνει

δουλειά, πιο συγκεκριμένα η ανάθεση γίνεται στα φύλλα που δεν είναι σίγουρο ότι έχουν την τιμή 1.

Φάση W3: αλλιώς ονομάζεται και φάση εργασίας. Στην φάση αυτή οι ενεργοί επεξεργαστές βρίσκονται ήδη στα φύλλα του δέντρου τα οποία πρέπει να γίνει δουλειά (δηλαδή να μετατραπούν τα 0 σε 1). Όλοι οι επεξεργαστές γράφουν την τιμή 1 στο φύλλο στο οποίο βρίσκονται ανατεθειμένοι.



W1: καταμέτρηση επεξεργαστών (υπερ-εκτίμηση)
W2: ισοζυγισμένη ανάθεση επεξεργαστών σε στοιχεία της λίστας
W3: φάση εργασίας
W4: αξιολόγηση προόδου (υπο-εκτίμηση)

Σχήμα 5.1 – Φάσεις αλγορίθμου W [4]

Φάση W4: ή αλλιώς φάση αξιολόγησης προόδου. Σε αυτή την φάση γίνεται καταμέτρηση της προόδου που έχει γίνει μετά την ολοκλήρωση της Φάσης W3. Οι επεξεργαστές ξεκινούν από τα φύλλα στα οποία ήδη βρίσκονται από την Φάση W3, και εφαρμόζοντας εκδοχή του αλγορίθμου παράλληλης άθροισης (για το μοντέλο PRAM) διαβαίνουν το δέντρο προόδου. Με την ολοκλήρωση της Φάσης W4, στην ρίζα του δέντρου προόδου βρίσκεται το συνολικό πλήθος των φύλλων που έχουν τιμή 1.

Για να τερματίσει ο βρόγχος που περιλαμβάνει τις τέσσερις αυτές φάσεις, πρέπει η ρίζα του δέντρου προόδου να έχει τιμή n.

5.3 Πολυπλοκότητα Αλγορίθμου W

Όπως έχει ήδη αποδειχθεί [4] η πολυπλοκότητα κόστους του Αλγορίθμου W είναι:

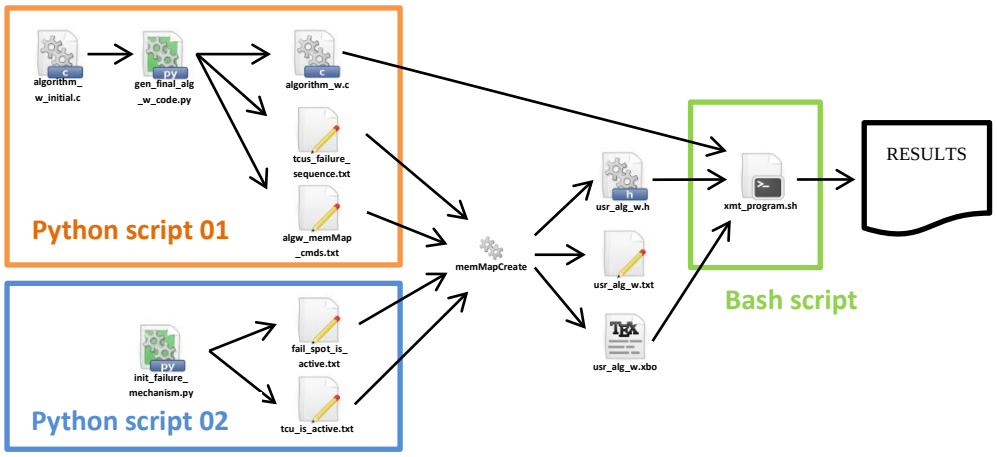
$$C_n(n, f) = O\left(\frac{n(\log_2 n)^2}{\log_2 \log_2 n}\right) \quad , \text{όπου } f < n$$

5.4 Μηχανισμός Πρόκλησης Σφαλμάτων

Η πλατφόρμα XMT δεν παρέχει την δυνατότητα πρόκλησης σφαλμάτων κατά την λειτουργία των επεξεργαστών. Για να είμαστε σε θέση να προκαλούμε σφάλματα κατά την εκτέλεση του αλγορίθμου έπρεπε να εισάγουμε ένα δικό μας μηχανισμό πρόκλησης σφαλμάτων. Αυτός ο μηχανισμός πρόκλησης σφαλμάτων έχει ενσωματωθεί επιτυχώς στην υλοποίηση του αλγορίθμου W. Πιο μετά θα επεξηγήσουμε τον ακριβές τρόπο λειτουργίας του μηχανισμού αυτού αλλά και το πώς μπορούμε να διαχειριστούμε τα σενάρια. Μια απλή λύση με flags δεν μας παρέχει την απαραίτητη ευελιξία για την δημιουργία συγκεκριμένων σεναρίων με μεταβλητό αριθμό σφαλμάτων. Η λύση με τα flags θα μας παρείχε την δυνατότητα μόνο για την πρόκληση σταθερού αριθμού σφαλμάτων σε συγκεκριμένο σημείο και σε συγκεκριμένη χρονική στιγμή στην εκτέλεση του αλγορίθμου. Προφανώς είναι αρκετά περιοριστικό για το εύρος των σεναρίων που μπορούμε να προσομοιώσουμε. Ενώ ο μηχανισμός πρόκλησης σφαλμάτων που ενσωματώσαμε μας επιτρέπει την επιλογή σημείου κατάρρευσης στον κώδικα και επιπλέον μας δίδει την δυνατότητα την επιλογή τρόπου κατάρρευσης επεξεργαστών (σταθερό ή μεταβλητό αριθμό καταρρεύσεων) και επιτρέπει την διαδοχική κατάρρευση επεξεργαστών.

Προτού αναφέρουμε λεπτομέρειες για τον τρόπο λειτουργίας του μηχανισμού πρόκλησης σφαλμάτων θα μελετήσουμε από το Σχήμα 5.2 διαγραμματικά την ροή των δεδομένων από την αρχή μέχρι την προσομοίωση του αλγορίθμου (το ίδιο σχήμα υπάρχει σε μεγαλύτερη ευκρίνεια στο Παράρτημα Β). Ο λόγος για τον οποίο συμβαίνει αυτό είναι γιατί με αυτό τον τρόπο θα κατανοήσουμε τη σημασία των μερών που συνθέτουν τον μηχανισμό πρόκλησης σφαλμάτων. Παρατηρείστε ότι στην περίπτωση του αλγορίθμου W γίνεται χρήση και του εργαλείου memMapCreate για την δημιουργία-αρχικοποίηση της μνήμης πριν την προσομοίωση. Σημαντικό στοιχείο είναι ότι πριν την χρήση του εργαλείου memMapCreate πρέπει να εκτελεστούν τα δύο python scripts. Σκοπός των δύο python scripts είναι να βοηθήσουν στην ρύθμιση και

ετοιμασία των επί μέρους αρχείων που είναι απαραίτητα τόσο στο compiling αλλά και στο simulation.



Σχήμα 5.2 – Ροή δεδομένων αλγορίθμου W

Πιο συγκεκριμένα ο μηχανισμός πρόκλησης σφαλμάτων αποτελείται από τα σημεία σφάλματος (fail spots), τα σημεία ελέγχου “if (tcu_is_active[$])” και τις δομές στην μνήμη (activation flags, tcus failure sequence, etc). Τα σημεία σφάλματος είναι στο σύνολο τους 29 και βρίσκονται στα μη παράλληλα σημεία του κώδικα xmtc. Ο λόγος για τον οποίο συμβαίνει αυτό είναι προφανές αφού έτσι μας δίδεται η δυνατότητα (όταν δεν εκτελείται κάποιο παράλληλο σημείο κώδικα) να προκαλέσουμε καταρρεύσεις επεξεργαστών. Κυρίως για λόγους απλοποίησης του μηχανισμού πρόκλησης σφαλμάτων δεν δώσαμε την δυνατότητα να υποστηρίζονται σφάλματα εντός παράλληλων σημείων του κώδικα. Αν αυτή η δυνατότητα ήταν ενσωματωμένη στον μηχανισμό πρόκλησης σφαλμάτων τότε θα αλλοιωνόταν σημαντικά η πολυπλοκότητα του αλγορίθμου. Τα 29 σημεία σφαλμάτων που υποστηρίζονται, ασυμπτωτικά δεν επηρεάζουν την πολυπλοκότητα του αλγορίθμου W σε τέτοιο βαθμό ώστε να αλλοιώνουν την ποιότητα των αποτελεσμάτων της προσομοίωσης.

Screenshot 5.1 – Μεταβολές κώδικα εξαιτίας "gen_final_alg_w_code.py"

5.4.1 Python script "gen_final_alg_w_code.py"

Το πρώτο python script ονομάζεται "gen_final_alg_w_code.py", αφού διαβάσει την αρχική μορφή του αλγορίθμου W πραγματοποιεί τις ακόλουθες διεργασίες:

- Τοποθετεί – παρεμβάλλει στον αρχικό κώδικα τις global μεταβλητές (n, log_n)
- Τοποθετεί – παρεμβάλλει στον αρχικό κώδικα, επιπλέον εντολές (κώδικα) για κάθε ένα από τα σημεία σφάλματος (fail spots)
- Τοποθετεί – παρεμβάλλει στον αρχικό κώδικα τους ελέγχους "if (tcu_is_active[$])" ούτως ώστε οι επεξεργαστές που καταρρέουν να μην λαμβάνουν μέρος πλέον στην ολοκλήρωση των εργασιών.
- Δημιουργεί μια τυχαία σειρά κατάρρευσης των επεξεργαστών και την αποθηκεύει στο αρχείο "tcus_failure_sequence.txt"
- Δημιουργεί και αποθηκεύει στο αρχείο "algw_memMap_cmds.txt" την κατάλληλη ακολουθία από εντολές, οι οποίες γίνονται fed με απλή ανακατεύθυνση στο εργαλείο memMapCreate αργότερα για την δημιουργία – δέσμευση μνήμης
- ΣΗΜΑΝΤΙΚΟ είναι το γεγονός ότι κατά την κλίση του script ο χρήστης μπορεί να καθορίσει σε ποιο mode θα εργάζεται το failure mechanism (υπάρχουν τα modes 1 και 2). Εάν δεν οριστεί η παράμετρος **-m** τότε το failure mechanism αυτόματα επιλέγει το mode 1. Στο mode 1 ο χρήστης μπορεί μέσω της

παραμέτρου **-f** να καθορίσει το πλήθος των επεξεργαστών που επιθυμεί να καταρρέουν σε κάθε fail spot (δηλαδή σταθερό αριθμό). Αντιθέτως στο mode 2 ο χρήστης μπορεί να καθορίσει μεταβλητό αριθμό σφαλμάτων σε κάθε fail spot μέσω της παραμέτρου **-p** (προς το παρόν μόνο φθίνουσα ακολουθία μπορεί να ακολουθείται).

Πρίν

(algorithm_w_initial.c)

```
93    // > > >   B E G I N   O F   M A I N   B O D Y   < < <
94
95    /// <> fail spot 1
96
97    //#########################################################################
98    //# initial work item for PID
99    //#########################################################################
100   spawn(1, max_tcu_num){
101   //< if tcu active
102       k[$] = $;   // k := PID; --inital work item for PID
103   //> endif tcu active
104   }
105   //-------------------------------------------------------------------------
106
107   /// <> fail spot 2
108
109   //#########################################################################
110   //# W3
111   //#########################################################################
112   spawn(1, max_tcu_num){
113   //< if tcu active
114       task[k[$]] = 1; // perform a task
115   //> endif tcu active
116   }
117   //-------------------------------------------------------------------------
```

Μετά

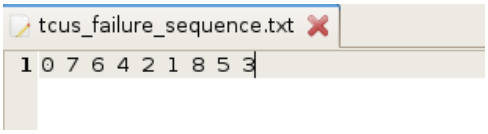
(algorithm_w.c)

```
94    // > > >   B E G I N   O F   M A I N   B O D Y   < < <
95
96    /// <> fail spot 1
97  if ( (fail_spot_is_active[1]) && (fail_index < n) ){
98      if ( fps_mode == 2 ){
99          number_of_failures_per_spot = ((n - fail_index) + 1) / fps_paronomastis;
100         if (number_of_failures_per_spot == 0)
101             number_of_failures_per_spot = 1;
102     }
103     for (tmp_count = 1; tmp_count <= number_of_failures_per_spot; tmp_count++){
104         if ( fail_index < n ){
105             tcu_is_active[tcus_failure_sequence[fail_index]] = 0;
106             fail_index++;
107         }else{
108             break;
109         }
110     }//end of "for" loop
111 }
112
113   //#########################################################################
114   //# initial work item for PID
115   //#########################################################################
116   spawn(1, max_tcu_num){
117   if ( tcu_is_active[$] ){//< if tcu active
118       k[$] = $;   // k := PID; --inital work item for PID
119   }//> endif tcu active
120   }
121   //-------------------------------------------------------------------------
122
123   /// <> fail spot 2
124 if ( (fail_spot_is_active[2]) && (fail_index < n) ){
125   if ( fps_mode == 2 ){
```

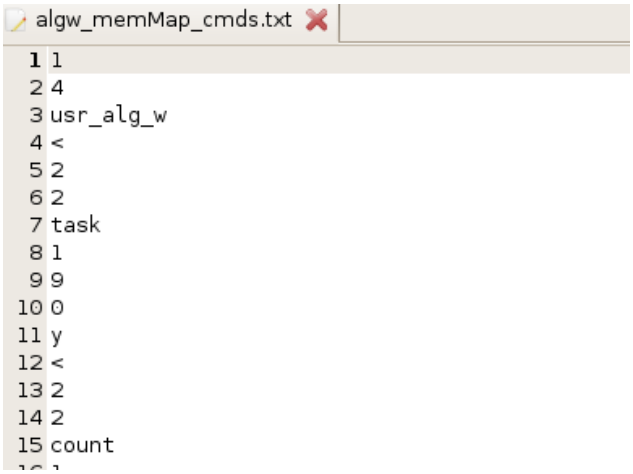
Screenshot 5.2 – Μεταβολές κώδικα εξαιτίας "gen_final_alg_w_code.py"

Στο Screenshot 5.1 και Screenshot 5.2 παρουσιάζονται οι αλλαγές που προκαλούνται στον κώδικα. Παρατηρείστε ότι ο παραγόμενος κώδικας αποθηκεύεται στο αρχείο algorithm_w.c ενώ το αρχείο algorithm_w_initial.c παραμένει αναλλοίωτο ούτος ώστε να επαναχρησιμοποιηθεί σε μετέπειτα εκτελέσεις του python script "gen_final_alg_w_code.py".

Επίσης το python script "gen_final_alg_w_code.py" παράγει τα αρχεία "tcus_failure_sequence.txt" και "algw_memMap_cmds.txt". Μια πιθανή όψη των αρχείων αυτών, είναι αυτή που παρουσιάζεται στο Screenshot 5.3 και Screenshot 5.4 (για $n = 8$). Προσοχή στο αρχείο "tcus_failure_sequence.txt" το πρώτο στοιχείο είναι μηδέν εξαιτίας του ότι ο πίνακας ορισμένος ως array[0…8].



Screenshot 5.3 – Πιθανή όψη αρχείου "tcus_failure_sequence.txt", για $(n = 8)$

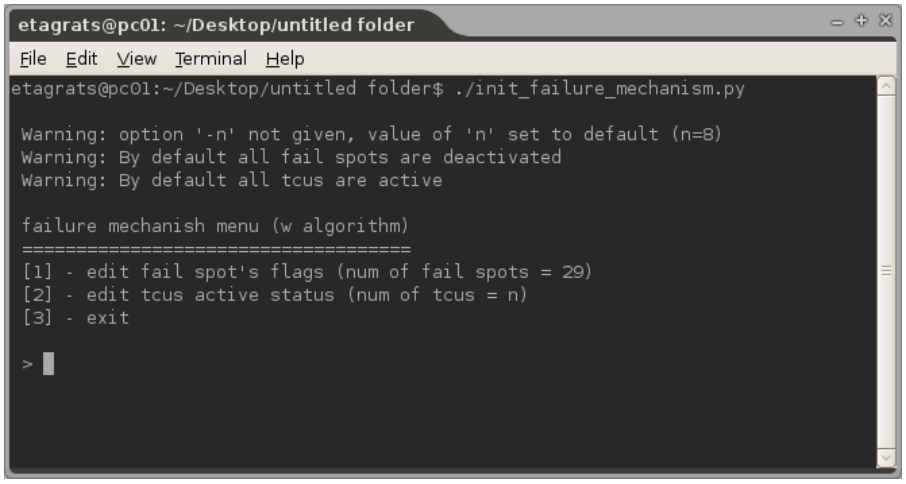


Screenshot 5.4 – Πιθανή όψη αρχείου "algw_memMap_cmds.txt", για $(n = 8)$

5.4.2 Python script "init_failure_mechanism.py"

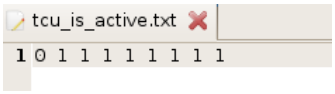
Το python script "init_failure_mechanism.py" έχει ως σκοπό του την διευκόλυνση της δημιουργίας σεναρίων κατάρρευσης των επεξεργαστών. Με το πέρας της εκτέλεσης του συγκεκριμένου script δημιουργούνται δύο αρχεία (fail_spot_is_active.txt και tcu_is_active.txt). Το αρχείο "fail_spot_is_active.txt" περιέχει τις τιμές των flags για

κάθε πιθανό σημείο σφάλματος. Δηλαδή πιο απλά οι τιμές στο αρχείο αυτό καθορίζουν ποιά fail spots είναι ενεργοποιημένα στον αλγόριθμο W και θα δημιουργήσουν σφάλματα. Το αρχείο "tcu_is_active.txt" καθορίζει το ποιοί επεξεργαστές είναι ενεργοποιημένοι κατά την εκκίνηση της προσομοίωσης του αλγορίθμου W.

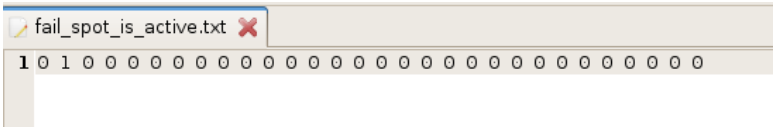


Screenshot 5.5 – Κυρίως οθόνη αρχικοποίησης μηχανισμού πρόκλησης σφαλμάτων

Επίσης το python script "init_failure_mechanism.py" παράγει τα αρχεία "tcu_is_active.txt" και "fail_spot_is_active.txt". Μια πιθανή όψη των αρχείων αυτών, είναι αυτή που παρουσιάζεται στο Screenshot 5.6 και Screenshot 5.7 (για $n = 8$). Προσοχή στο αρχείο "tcu_is_active.txt" το πρώτο στοιχείο είναι μηδέν εξαιτίας του ότι ο πίνακας είναι ορισμένος array[0…8]. Στο αρχείο "fail_spot_is_active.txt" το πρώτο στοιχείο είναι μηδέν εξαιτίας του ότι ο πίνακας ορίζεται ως array[0…29].



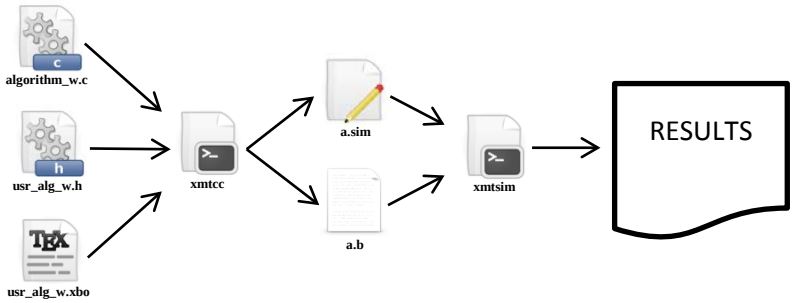
Screenshot 5.6 – Πιθανή όψη αρχείου "tcu_is_active.txt", για $(n = 8)$



Screenshot 5.7 – Πιθανή όψη αρχείου "fail_spot_is_active.txt"

5.4.3 Bash script “xmt_program.sh”

Το bash script “xmt_program.sh” έχει ως σκοπό του την αυτοματοποίηση του compiling και simulation του αλγορίθμου W. Το Σχήμα 5.3 περιγράφει πιο αναλυτικά το compiling και το simulation (το ίδιο σχήμα υπάρχει σε μεγαλύτερη ευκρίνεια στο Παράρτημα Β). Όπως φαίνεται γίνεται σε δύο φάσεις. Κατά την πρώτη φάση τα αρχεία “algorithm_w.c”, “usr_alg_w.h” και “usr_alg_w.xbo” γίνονται compile με την χρήση του εργαλείου xmtcc. Ακολούθως τα αρχεία εξόδου από την πρώτη φάση γίνονται δεδομένα εισόδου για την δεύτερη φάση. Κατά την δεύτερη φάση χρησιμοποιώντας το εργαλείο xmtsim γίνεται προσομοίωση του προγράμματος.



Σχήμα 5.3 – Διαγραμματική απεικόνιση του compiling και simulation

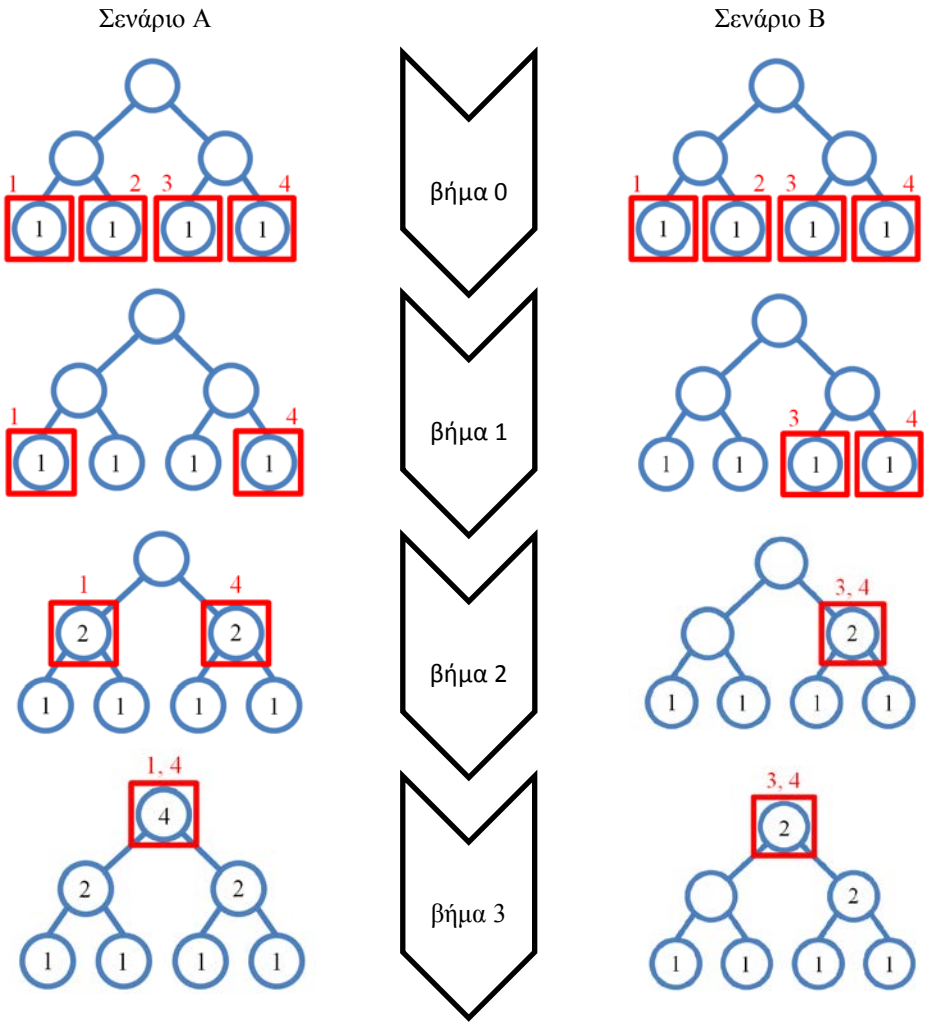
5.5 Ομάδες Σεναρίων για Προσομοίωση

Για την επιτυχημένη αξιολόγηση του υλοποιημένου αλγορίθμου W, πρέπει να ετοιμαστούν στοχευμένα σενάρια τα οποία να δοκιμάζουν τον αλγόριθμο στα όρια του. Η ποικιλία στο πλήθος των σεναρίων θα οδηγήσει σε ασφαλέστερη εξαγωγή συμπερασμάτων.

Στην συνέχεια αυτού του υποκεφαλαίου θα παρουσιαστούν τα σενάρια που θα τρέξουν στον προσομοιωτή. Προκειμένου να κατανοήσουμε ευκολότερα τα σενάρια, θα τα ομαδοποιήσουμε με σκοπό να αποφύγουμε το μπέρδεμα (λόγο του μεγάλου αριθμού σεναρίων).

Να τονίσουμε ότι για όλες τις ομάδες σεναρίων (πλην της πρώτης ομάδας - υποκεφάλαιο 5.5.1) γίνονται πολλαπλές προσομοιώσεις και λήψεις μετρήσεων γιατί η κατάρρευση των επεξεργαστών γίνεται τυχαία. Επομένως οι επεξεργαστές ανατίθενται σε διαφορετικά φύλλα κάθε φορά με αποτέλεσμα να επηρεάζεται ο χρόνος ολοκλήρωσης. Χαρακτηριστικά θα εξηγήσουμε ένα απλό παράδειγμα το οποίο δεν

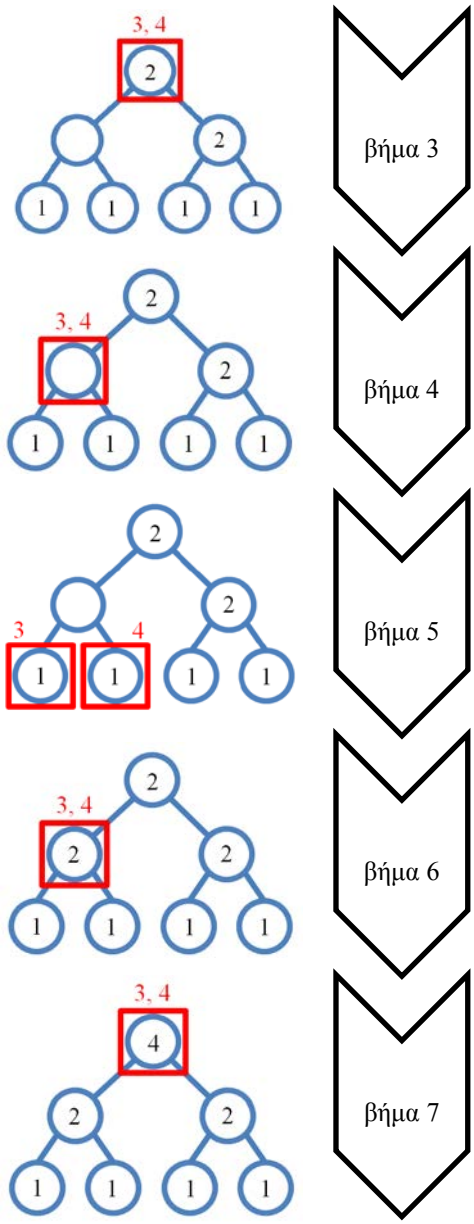
ανήκει σε κάποια από τις ομάδες, αλλά ο σκοπός του είναι να μας διδάξει το γιατί διαφορετικές εκτελέσεις του ίδιου σεναρίου έχουν διαφορετικό χρόνο ολοκλήρωσης. Στο Σχήμα 5.4 παρουσιάζονται δύο πιθανές προσομοιώσεις για πρόβλημα μεγέθους $n = 4$ και πλήθος σφαλμάτων $f = {}^{n}/_{2}$ κατά την πρώτη εκτέλεση της φάσης W4 (γραμμή 3 - Ψευδοκώδικας 5.1). Πιο συγκεκριμένα τα σφάλματα γίνονται αμέσως μετά που οι επεξεργαστές γράφουν την τιμή 1 στα φύλλα (και στο δέντρο προόδου), και πριν ξεκινήσουν την διάσχιση του δέντρου προόδου.



Σχήμα 5.4 – Πιθανά σενάρια εκτέλεσης

Επειδή καταρρέουν τυχαία οι επεξεργαστές υπάρχει το ενδεχόμενο να συμβούν τα σενάρια Α και Β. Αυτό που θα τονίσουμε είναι το τι συμβαίνει σε κάθε περίπτωση. Προσέξτε ότι στο ‘βήμα 0’ όλοι οι επεξεργαστές και στις δύο περιπτώσεις γράφουν την τιμή 1 στα φύλλα και στο δέντρο προόδου. Με την κατάρρευση των επεξεργαστών,

επηρεάζετε το αποτέλεσμα κατά την φάση της αξιολόγησης προόδου (φάση W_4 – χρήση δέντρου προόδου). Συγκεκριμένα προσέξτε ότι στο σενάριο Α, επειδή υπάρχει ένας επεξεργαστής σε κάθε υπόδεντρο της ρίζας, είναι δυνατή η καταμέτρηση - ανίχνευση της δουλειάς που έχουν ολοκληρώσει και οι επεξεργαστές με σφάλμα. Αντιθέτως στο σενάριο Β, και οι δύο επεξεργαστές που παραμένουν ενεργοί βρίσκονται στο ίδιο υπόδεντρο της ρίζας, με αποτέλεσμα να μην λαμβάνεται υπόψη στο τελικό σύνολο η εργασία των επεξεργαστών με σφάλμα. Σημαντικό είναι το γεγονός ότι στο σενάριο Α χρειάστηκαν μόνο 3 βήματα για την ολοκλήρωση του αλγορίθμου, ενώ στο σενάριο Β χρειάζονται κάποια βήματα ακόμη. Τα επιπλέον βήματα που απαιτούνται παρουσιάζονται εν συντομία στο Σχήμα 5.5.



Σχήμα 5.5 - Επιπλέον βήματα για ολοκλήρωση σεναρίου Β (συνέχεια από Σχήμα 5.4)

Όπως φαίνεται και από τα βήματα στο Σχήμα 5.5, πρέπει οι επεξεργαστές να ανατεθούν στο αριστερό υπόδεντρο της ρίζας προκειμένου να ολοκληρώσουν τη δουλειά που ο αλγόριθμος θεωρεί ότι δεν έχει γίνει (παρόλο που οι επεξεργαστές 1 και 2 έγραψαν την τιμή 1 στα φύλλα – υποεκτίμηση). Αφού οι επεξεργαστές τελικά ανατεθούν στα φύλλα ολοκληρώνουν την εργασία και στην συνέχεια εκτελούν αξιολόγηση προόδου και τελικά ολοκληρώνεται η εκτέλεση.

Τα δύο σενάρια που μελετήσαμε τονίζουν την ανάγκη για εκτέλεση πολλαπλών προσομοιώσεων ούτως ώστε τα αποτελέσματα των μετρήσεων να είναι πιο έγκυρα.

5.5.1 Ομάδα Σεναρίων Χωρίς Σφάλματα

Στην ομάδα αυτή ανήκει μόνο ένα σενάριο, στο οποίο ΔΕΝ υπάρχουν σφάλματα.

5.5.2 Ομάδα Σεναρίων Υπερεκτίμησης

Σε αυτή την ομάδα σεναρίων ανήκουν οι προσομοιώσεις οι οποίες ελέγχουν την συμπεριφορά του αλγορίθμου W σε περιπτώσεις υπερεκτιμήσεων. Δηλαδή κατά την διάρκεια εκτέλεσης του αλγορίθμου W γίνονται διαδοχικές υπερεκτιμήσεις ως προς τον αριθμό ενεργών επεξεργαστών. Για την επίτευξη αυτού του είδους σεναρίων γίνεται ενεργοποίηση των σημείων πρόκλησης σφαλμάτων ‘fail spot 1’ και ‘fail spot 15’. Επίσης να τονίσουμε ότι χρησιμοποιείται το mode 2 του μηχανισμού πρόκλησης σφαλμάτων και προσομοιώνονται σενάρια για τιμές του $p \in P = \{ 2,\ 4 \}$, όπου p ο συντελεστής του μηχανισμού σφάλματος.

5.5.3 Ομάδα Σεναρίων Υποεκτίμησης

Σε αυτή την ομάδα σεναρίων ανήκουν οι προσομοιώσεις οι οποίες ελέγχουν την συμπεριφορά του αλγορίθμου W σε περιπτώσεις υποεκτιμήσεων. Δηλαδή κατά την διάρκεια εκτέλεσης του αλγορίθμου W γίνονται διαδοχικές υποεκτιμήσεις ως προς τον συνολικό όγκο της δουλειάς που έχει διεκπεραιωθεί. Για την επίτευξη αυτού του είδους σεναρίων γίνεται ενεργοποίηση των σημείων πρόκλησης σφαλμάτων ‘fail spot 3’ και ‘fail spot 25’. Επίσης να τονίσουμε ότι χρησιμοποιείται το mode 2 του μηχανισμού πρόκλησης σφαλμάτων και προσομοιώνονται σενάρια για τιμές του $p \in P = \{ 2,\ 4 \}$, όπου p ο συντελεστής του μηχανισμού σφάλματος.

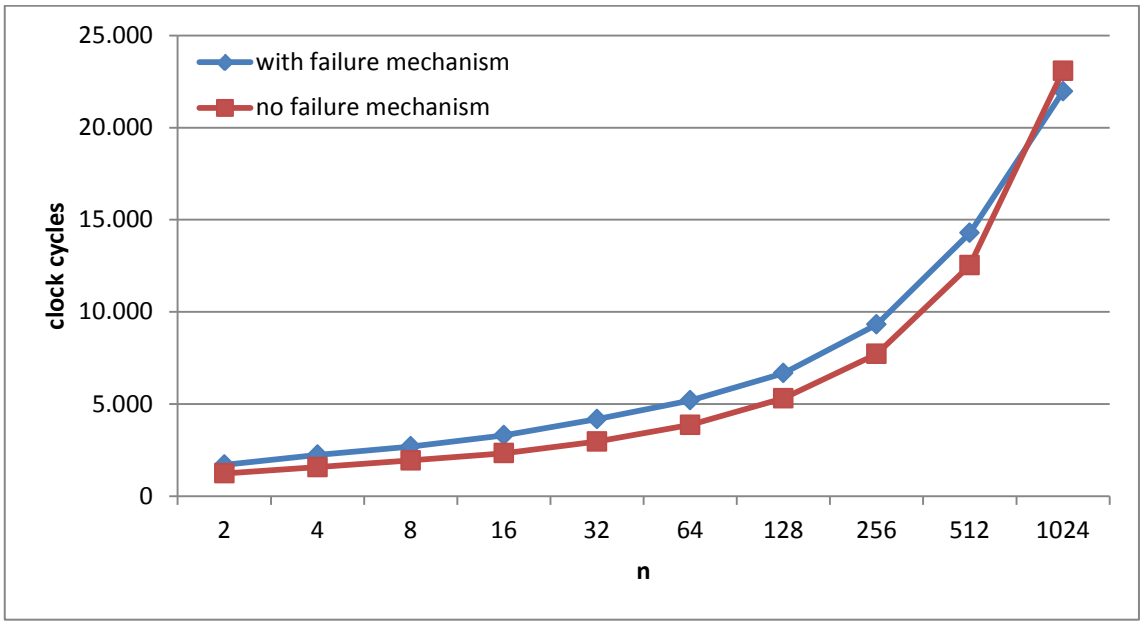
5.6 Πειραματική Αξιολόγηση και Μετρικές

Αρχικά έγιναν δύο βασικές προσομοιώσεις οι οποίες είχαν ως σκοπό να δείξουν ότι η προσθήκη του μηχανισμού πρόκλησης σφαλμάτων δεν επηρεάζει σημαντικά τον χρόνο ολοκλήρωσης του αλγορίθμου. Το σενάριο το οποίο προσομοίωναν οι δύο υλοποιήσεις ήταν απλή εκτέλεση χωρίς σφάλματα. Η διαφορά των δύο είναι το γεγονός ότι η υλοποίηση ‘no failure mechanism’ δεν συμπεριλάμβανε τον μηχανισμό πρόκλησης σφαλμάτων, ενώ η υλοποίηση ‘with failure mechanism’ αποτελεί τροποποίηση της πρώτης με προσθήκη του μηχανισμού πρόκλησης σφαλμάτων. Πίνακες με λεπτομερείς μετρήσεις υπάρχουν στο Παράρτημα Β. Στην συνέχεια (Πίνακας 5.1) παρουσιάζονται

οι μετρήσεις για τις δύο αυτές προσομοιώσεις. Η γραφική αναπαράσταση των μετρήσεων αυτών φαίνεται στο Γράφημα 5.1.

n	with failure mechanism	no failure mechanism
2	1.710	1.246
4	2.252	1.584
8	2.699	1.945
16	3.312	2.342
32	4.186	2.972
64	5.203	3.878
128	6.681	5.318
256	9.314	7.720
512	14.283	12.529
1024	21.961	23.068

Πίνακας 5.1 – Clock cycles ανά υλοποίηση



Γράφημα 5.1 – Clock cycles ανά υλοποίηση (with failure mechanism / no failure mechanism)

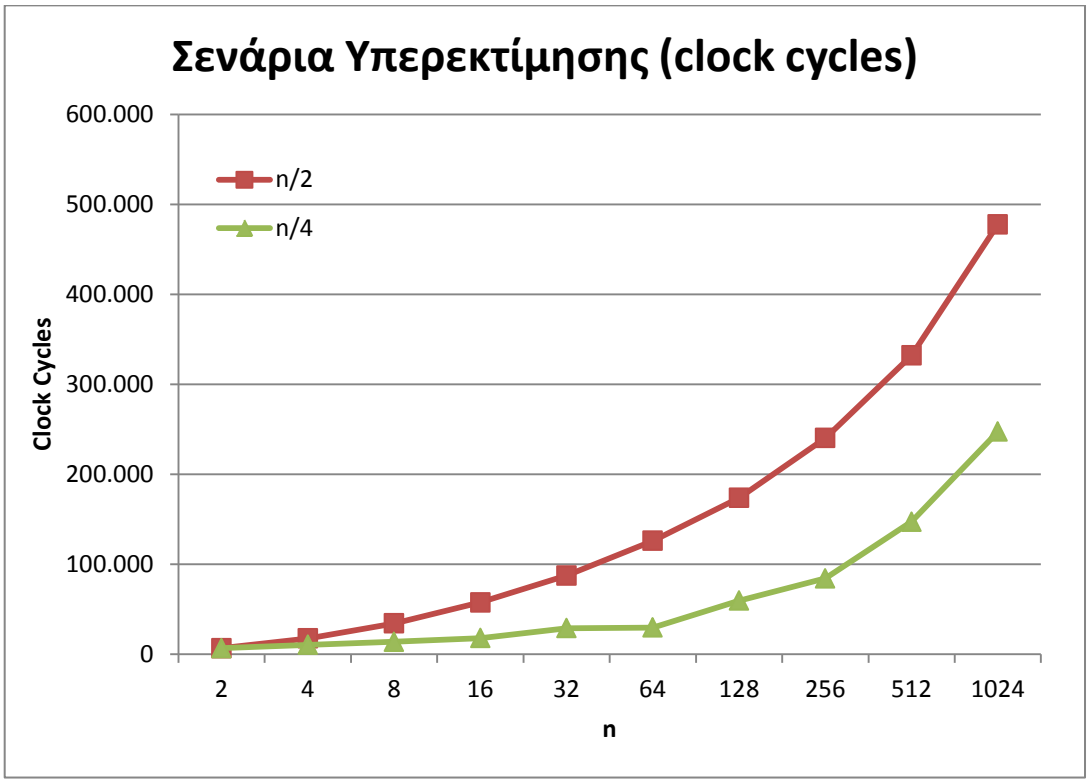
Η διαφορά που παρουσιάζεται για μεγάλες τιμές του n στο γράφημα οφείλεται στο ότι η πλατφόρμα XMT βρίσκεται στα όρια της. Επίσης παρατηρούμε ότι ο συνολικός χρόνος ολοκλήρωσης δεν επηρεάζεται σημαντικά (μάλιστα μπορούμε να πούμε ότι ασυμπτωτικά είναι οι ίδιοι).

Στην συνέχεια ακολουθούν οι μετρήσεις που λήφθηκαν για την ομάδα σεναρίων υπερεκτίμησης. Ο Πίνακας 5.2 παρουσιάζει τις μετρήσεις ανά σενάριο, ενώ το Γράφημα 5.2 την γραφική τους αναπαράσταση. Αξίζει να παρατηρήσουμε ότι ανάλογα με το πλήθος των επεξεργαστών που καταρρέουν σε κάθε επανάληψη, αυξάνεται ο

συνολικός χρόνος ολοκλήρωσης. Δηλαδή όταν μεγαλώνει το πλήθος των σφαλμάτων αυξάνεται ο χρόνος ολοκλήρωσης, ενώ όταν μειώνεται το πλήθος των σφαλμάτων ταυτόχρονα μειώνεται και ο συνολικός χρόνος ολοκλήρωσης. Κάλλιστα μπορούμε να πούμε ότι το πλήθος των σφαλμάτων καθορίζει το μέγιστο δυνατό παραγόμενο έργο σε κάθε επανάληψη του αλγορίθμου.

n	f=n/2	f=n/4
2	6.743	6.747
4	17.658	10.473
8	34.327	13.915
16	57.686	17.931
32	87.627	28.947
64	126.060	29.746
128	173.804	59.559
256	240.238	84.316
512	332.067	147.241
1024	477.692	247.515

Πίνακας 5.2 – Clock cycles ανά σενάριο υπερεκτίμησης



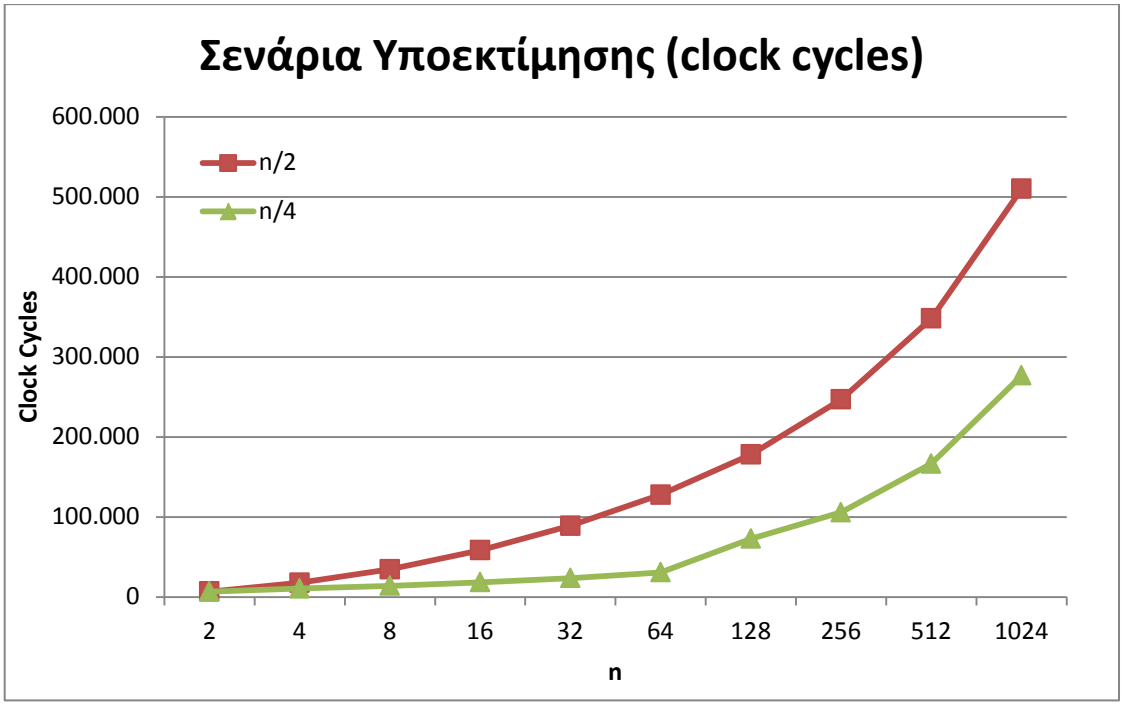
Γράφημα 5.2 – Γραφική αναπαράσταση μετρήσεων για τα σενάρια υπερεκτίμησης

Στον πίνακα που ακολουθεί (Πίνακας 5.3) παρουσιάζονται οι μετρήσεις που λήφθηκαν κατά την προσομοίωση των σεναρίων υποεκτίμησης. Συγκεκριμένα τα σενάρια αυτά

έχουν παρόμοια συμπεριφορά με τα σενάρια υπερεκτίμησης. Δηλαδή αυξημένος αριθμός επεξεργαστών που καταρρέουν σε κάθε επανάληψη συνεπάγεται και αυξημένος χρόνος ολοκλήρωσης. Παρομοίως μειωμένος αριθμός επεξεργαστών που καταρρέουν οδηγεί σε πιο σύντομη ολοκλήρωση του αλγορίθμου.

n	n/2	n/4
2	6.760	6.761
4	17.685	10.472
8	34.443	13.969
16	58.255	18.235
32	88.967	23.493
64	127.697	30.652
128	178.059	72.765
256	246.728	105.798
512	348.113	166.529
1024	510.062	276.830

Πίνακας 5.3 - Clock cycles ανά σενάριο υποεκτίμησης



Γράφημα 5.3 - Γραφική αναπαράσταση μετρήσεων για τα σενάρια υποεκτίμησης

Επίσης λήφθηκαν μετρήσεις για να υπολογιστεί το κόστος του αλγορίθμου για τα σενάρια υπερεκτίμησης και υποεκτίμησης. Ο Πίνακας 5.4 παρουσιάζει το κόστος του

αλγορίθμου για τα σενάρια υπερεκτίμησης και υποεκτίμησης καθώς επίσης και το θεωρητικό κόστος του αλγορίθμου. Το θεωρητικό κόστος [4] υπολογίζεται ως εξής:

$$C_n(n,f) = O\left(\frac{n * (\log_2 n)^2}{\log_2 \log_2 n}\right)$$

Ο υπολογισμός του κόστους [4] για τα σενάρια υπερεκτίμησης και υποεκτίμησης έγινε ως εξής:

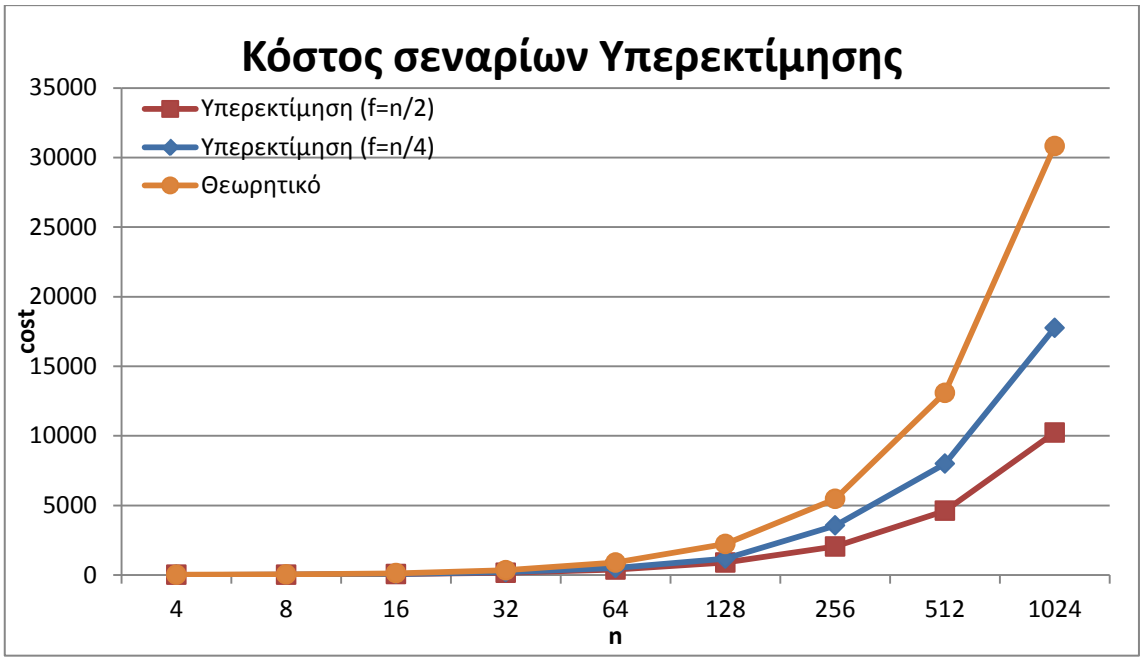
$$C_n(n,f) = \left[\sum_{\tau=1}^{\beta} \pi_\tau\right] * \log_2 n$$

, όπου β ο αριθμός των βημάτων για την επίλυση ενός προβλήματος μεγέθους n. Όπου π_τ είναι ο αριθμός των επεξεργαστών που δεν έχουν καταρρεύσει μέχρι το τέλος του βήματος τ.

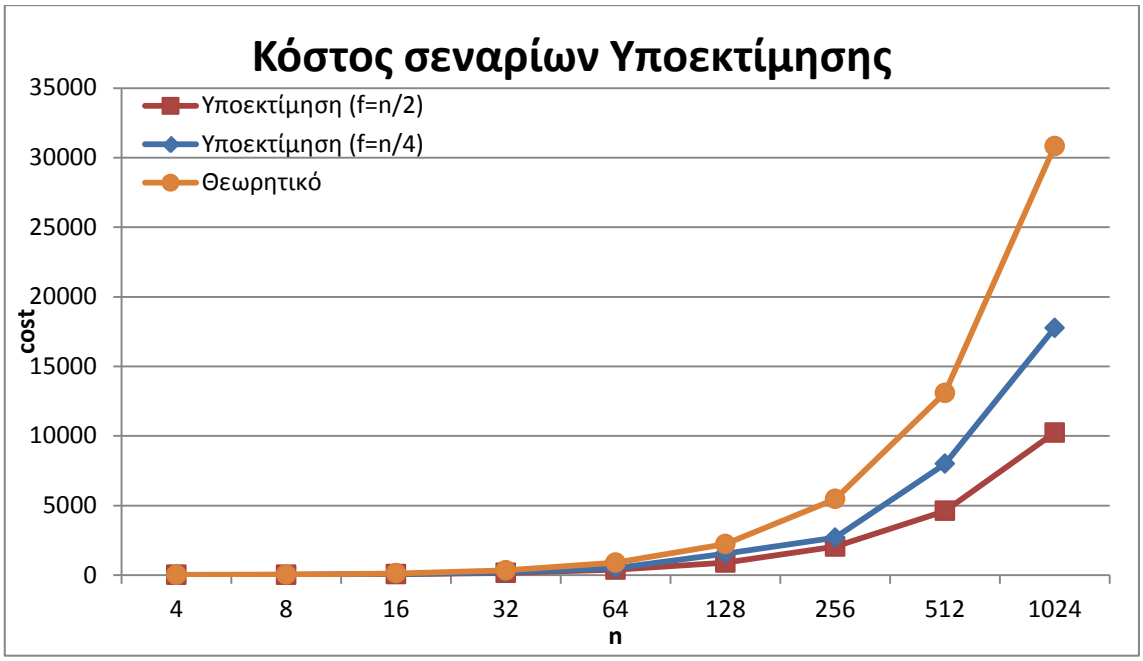
n	Υπερεκτίμηση		Υποεκτίμηση		Θεωρητικό
	f=n/2	f=n/4	f=n/2	f=n/4	
4	8	10	8	10	16,00
8	24	33	24	33	45,43
16	64	84	64	84	128,00
32	160	210	160	210	344,54
64	384	504	384	504	891,31
128	896	1176	896	1554	2234,13
256	2048	3552	2048	2688	5461,33
512	4608	7992	4608	7992	13082,96
1024	10240	17760	10240	17760	30825,47

Πίνακας 5.4 – Κόστος Αλγορίθμου W

Γραφική αναπαράσταση των μετρήσεων για τα σενάρια φαίνεται στο Γράφημα 5.4 και Γράφημα 5.5. Από τα γραφήματα παρατηρούμε ότι το θεωρητικό κόστος είναι χειρότερο από τα σενάρια υπερεκτίμησης και υποεκτίμησης. Ο λόγος για τον οποίο συμβαίνει αυτό είναι γιατί το θεωρητικό κόστος λαμβάνει υπόψη το χειρότερο σενάριο εκτέλεσης. Επίσης παρατηρούμε ότι η συμπεριφορά του κόστους για τα σενάρια ακολουθεί την ίδια συμπεριφορά με τα θεωρητικά δεδομένα. Αξίζει να τονίσουμε ότι και για τις δύο ομάδες σεναρίων (υπερεκτίμησης και υποεκτίμησης) το κόστος παραμένει στα ίδια επίπεδα, αυτό φαίνεται και από τις τιμές για τα κόστη (Πίνακας 5.4). Αναλυτικά όλες οι μετρήσεις που οδήγησαν στην δημιουργία του πιο πάνω πίνακα παρουσιάζονται στο Παράρτημα Β.



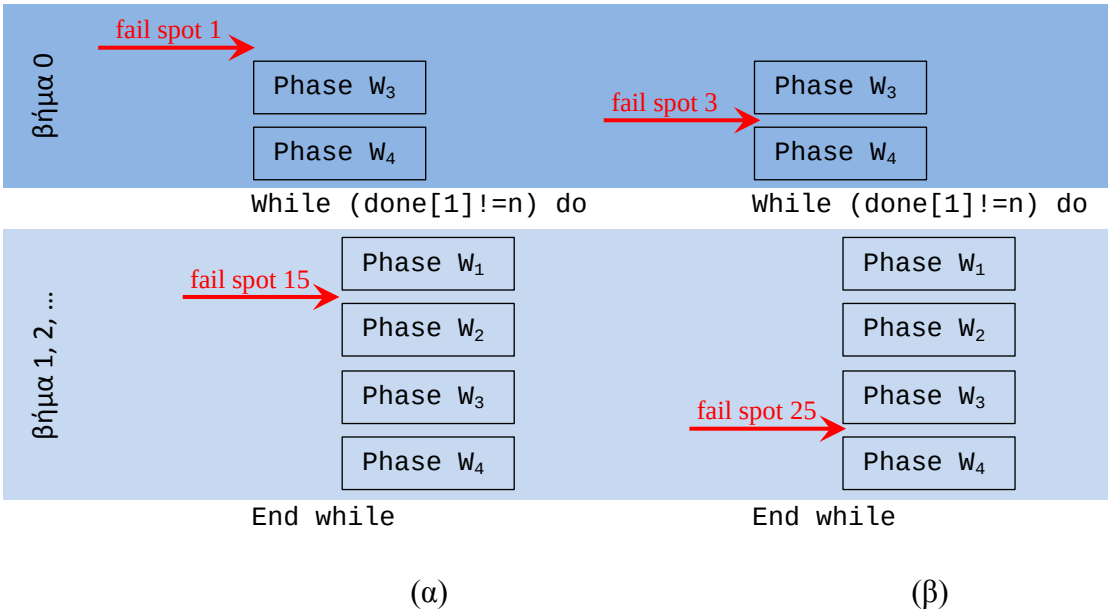
Γράφημα 5.4 – Κόστος σεναρίων Υπερεκτίμησης



Γράφημα 5.5 – Κόστος σεναρίων Υπερεκτίμησης

Το κόστος ήταν αναμενόμενο να είναι στα ίδια επίπεδα αφού σε κάθε βήμα του αλγορίθμου το πλήθος των σφαλμάτων είναι το ίδιο και για τα δύο σενάρια. Στο Σχήμα 5.6 παρουσιάζεται η οπτική απεικόνιση των σημείων πρόκλησης σφαλμάτων για τα δύο σενάρια. Παρατηρείστε από το σχήμα ότι εντός του κάθε βήματος υπάρχει μόνο ένα

σημείο σφάλματος. Για παράδειγμα τα σενάρια υπερεκτίμησης και υποεκτίμησης με f=n/2 σε κάθε βήμα καταρρέει ο μισός αριθμός από τους ενεργούς επεξεργαστές, επομένως κατά τον υπολογισμό του κόστους το πλήθος των επεξεργαστών σε κάθε βήμα θα είναι το ίδιο και για τα δύο σενάρια, συνεπώς το κόστος θα είναι στα ίδια επίπεδα.



Σχήμα 5.6 – Σημεία πρόκλησης σφαλμάτων για σενάρια (α) υπερεκτίμησης και (β) υποεκτίμησης.

Συνοψίζοντας τα αποτελέσματα, παρατηρούμε ότι η συμπεριφορά την πλατφόρμας XMT στην προσομοίωση του αλγορίθμου W ακολουθεί τα θεωρητικά πρότυπα. Πιο συγκεκριμένα ο χρόνος ολοκλήρωσης του αλγορίθμου ασυμπτωτικά είναι ο ίδιος με τον θεωρητικό. Επίσης παρατηρήσαμε ότι το κόστος της προσομοίωσης για τα σενάρια υπερεκτίμησης και υποεκτίμησης ακολουθεί όμοια συμπεριφορά με το θεωρητικό κόστος.

Κεφάλαιο 6

Αλγόριθμος X

6.1 Το Πρόβλημα Write-All στο A-PRAM

Το πρόβλημα Write-All [4] στο A-PRAM ορίζεται ως εξής: Δεδομένου μιας λίστας n στοιχείων, όπου αρχικά όλα τα στοιχεία έχουν την τιμή 0, ζητείται να μετατραπούν οι τιμές των στοιχείων σε 1 χρησιμοποιώντας ρ ασυγχρώνιστους επεξεργαστές στο μοντέλο A-PRAM.

6.2 Περιγραφή Αλγορίθμου X

Η κλασσική λύση για το πρόβλημα Write-All μεγέθους n με την χρήση ρ επεξεργαστών, όπου $\rho = n$, έχει κόστος $\Theta(n^2)$. Ο Αλγόριθμος X [4] [1] μπορεί να λύσει το ίδιο πρόβλημα με κόστος $C'_n(n) = O\left(n^{\log_2 3}\right) = O(n^{1,6})$. Επειδή στο A-PRAM οι επεξεργαστές είναι ασυγχρόνιστοι χρειάζεται ο αλγόριθμος να έχει κάποιο βαθμό ελευθερίας. Αυτό επιτυγχάνεται με την χρήση ενός δυαδικού δέντρου προόδου με n φύλλα το οποίο βρίσκεται στην κοινόχρηστη μνήμη, ο ρόλος του οποίου είναι να πληροφορεί τους επεξεργαστές για την πρόοδο (την ολοκλήρωση της εργασίας) που σημειώνεται. Επίσης χρησιμοποιείται ένας μονοδιάστατος πίνακας μεγέθους n, του οποίου τα κελιά αντιπροσωπεύουν τις θέσεις μνήμης που πρέπει να μετατραπούν από 0 σε 1. Ο Ψευδοκώδικας 6.1 παρουσιάζει τον αλγόριθμο X.

Πιο συγκεκριμένα ο αλγόριθμος ξεκινά με τους επεξεργαστές ανατεθειμένους στα φύλλα του δέντρου (line 06). Κάθε επεξεργαστής εκτελεί ένα βρόγχο (line 07-27) έως

ότου η τιμή στην ρίζα του δυαδικού δέντρου προόδου (done[1]) γίνει ίση με ένα. Εντός του βρόγχου εκτελούνται φωλιασμένοι έλεγχοι οι οποίοι καθορίζουν το ουσιαστικό κομμάτι του αλγορίθμου. Υπάρχουν τρείς βασικοί έλεγχοι (line 09, 10 και 14) οι οποίοι καθορίζουν τη θέση ενός επεξεργαστή στο δυαδικό δέντρο προόδου. Οι τρείς δυνατές θέσεις είναι (i) ο επεξεργαστής βρίσκεται σε κάποιο τυχαίο κόμβο του δέντρου και έχει ολοκληρώσει όλη τη δυνατή εργασία που μπορεί να κάνει στον συγκεκριμένο κόμβο, (ii) ο επεξεργαστής βρίσκεται σε κάποιο από τα φύλλα του δέντρου και υπάρχει δουλειά που πρέπει να γίνει και (iii) ο επεξεργαστής βρίσκεται σε κάποιον από τους εσωτερικούς κόμβους του δέντρου προόδου (δηλαδή όχι σε φύλλα) και υπάρχει δουλειά που πρέπει να ολοκληρωθεί. Αφαιρετικά παρατηρούμε ότι οι τρείς αυτοί βασικοί έλεγχοι κατηγοριοποιούν τις δυνατές θέσεις που μπορούν να βρεθούν οι επεξεργαστές ανά πάσα στιγμή. Να σημειώσουμε ότι σε κάθε μια από τις τρείς περιπτώσεις εκτελείται ένα συγκεκριμένο σύνολο εντολών. Για λόγους συντομίας θα αρκεστούμε σε μια απλή περιγραφή του πώς ο αλγόριθμος μεταχειρίζεται την κάθε περίπτωση ξεχωριστά.

Στην περίπτωση που ένας επεξεργαστής βρεθεί στην δεύτερη περίπτωση (ii), (βλέπε γραμμές 11-13) τότε σημαίνει ότι βρίσκεται σε φύλλο του δέντρου προόδου. Ένας επεξεργαστής σε αυτή τη θέση μπορεί να εκτελέσει εργασία και να ενημερώσει το δέντρο προόδου. Όμως αυτό γίνεται σε δύο ξεχωριστές επαναλήψεις ώστε να ικανοποιείται και η περίπτωση που κάποιοι επεξεργαστές καταρρεύσουν αφού ολοκληρώσουν την εργασία τους (και δεν φτάσουν να ενημερώσουν το δέντρο προόδου). Δηλαδή με πιο απλά λόγια, διασφαλίζεται ότι κάθε εργασία στα φύλλα εκτελείται επιτυχώς μόνο μία φορά.

Υπάρχει επίσης το ενδεχόμενο κάποιος επεξεργαστής να βρίσκεται σε ένα τυχαίο εσωτερικό κόμβο του δέντρου προόδου (όχι σε φύλλα), δηλαδή ανήκει στην περίπτωση (iii), δες γραμμές 15-25. Σε αυτή την περίπτωση γίνεται έλεγχος για την δουλειά που έχει ολοκληρωθεί στο αριστερό και δεξί παιδί του κόμβου. Στην συνέχεια βάση του ελέγχου που προαναφέραμε αποφασίζεται η μετακίνηση του επεξεργαστή στο αριστερό ή δεξί παιδί αναλόγως.

```
PRAM Pseudo-Code

LN
01  Processors i = 1 to n parbegin:
02    shared integer array task[1..n];       --the Write-All array, initially all values zero
03    shared integer array done[1..2n-1];    --the progress tree, initially all values zero
04    private integer where, d;              --current node index and its "done" value
05    private integer left, right;           --left/right child values in progress tree
06    where:=n+PID-1; --initial assignment at the leafs
07    while (done[1]≠1) do; --as long as the root is not marked as done
08      d:=done[where];
09      if (d=1) then where:=where div 2; --move up one level
10      elseif ( (d≠1)AND(where>n-1) ) then --at a leaf
11        if (task[where-n+1]=0) then task[where-n+1]:=1; --perform task
12        elseif (task[where-n+1]=1) then done[where]:=1; --indicate done
13        fi
14      elseif ( (d≠1)AND(where≤n-1) ) then --interior tree node
15        left:=done[2*where]; --left child value
16        right:= done[2*where+1]; --right child value
17        if ( (left=1)AND(right=1) ) then done[where]:=1; --both children done
18        elseif ( (left=0)AND(right=1) ) then where:=2*where; --move left
19        elseif ( (left=1)AND(right=0) ) then where:=2*where+1; --move right
20        elseif ( (left=0)AND(right=0) ) then --both subtrees are not done
21          --move down according to the PID bit
22          if ( PID[log(where)]=0 ) then where:=2*where; --move left
23          elseif ( PID[log(where)]=1 ) then where:=2*where+1; --move right
24          fi
25        fi
26      fi
27    od
28  parend
```

Ψευδοκώδικας 6.1 – Αλγόριθμος X [1]

Σκόπιμα αφήσαμε την περίπτωση (i) τελευταία στην ανάλυση του αλγορίθμου. Αυτό γιατί τώρα είμαστε σε θέση να αναγνωρίσουμε την πραγματική αξία της. Μέχρι αυτό το σημείο εξετάσαμε τις περιπτώσεις που ένας επεξεργαστής βρίσκεται στα φύλλα (περίπτωση ii, lines 11-13) ή σε κάποιο τυχαίο εσωτερικό κόμβο (περίπτωση iii, lines 15-25). Στην περίπτωση (ii) οι επεξεργαστές παραμένουν στα φύλλα ενώ στην περίπτωση (iii) μελετήσαμε το πώς οι επεξεργαστές μετακινούνται από τον οποιοδήποτε κόμβο πατέρα στα παιδιά. Αυτό που δεν μελετήσαμε είναι το πώς οι επεξεργαστές ανεβαίνουν επίπεδα (από τα παιδιά στον πατέρα). Το κενό αυτό έρχεται να συμπληρώσει η περίπτωση (i), δες γραμμή 9. Αυτό που συμβαίνει είναι το γεγονός ότι όταν ο κόμβος του δέντρου προόδου στον οποίο βρίσκεται ο επεξεργαστής αποκτήσει τιμή 1, τότε υπολογίζεται ο πατέρας του κόμβου αυτού και ο επεξεργαστής

μετακινείτε σε αυτόν. Αυτή η διαδικασία ονομάζεται και ανέβασμα επιπέδου. Στο παρόν σημείο ολοκληρώσαμε την ανάλυση του τρόπο λειτουργίας του αλγορίθμου Χ.

6.3 Πολυπλοκότητα Αλγορίθμου Χ

Όπως έχει ήδη αποδειχθεί η πολυπλοκότητα κόστους του Αλγορίθμου Χ είναι:

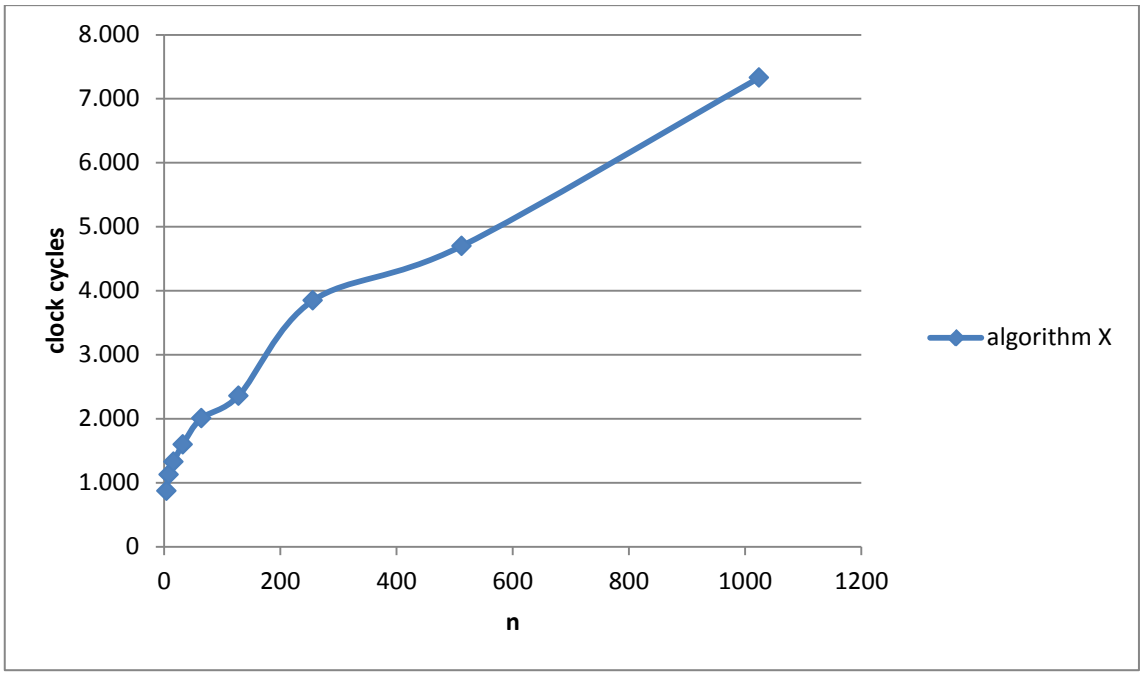
$$C'_n(n) = O\left(n^{\log_2 3}\right) = O(n^{1.6})$$

6.4 Πειραματική Αξιολόγηση και Μετρικές

Ο αλγόριθμος Χ υλοποιήθηκε και προσομοιώθηκε στην πλατφόρμα XMT. Είχαμε αποπειραθεί να υλοποιήσουμε και μηχανισμό πρόκλησης σφαλμάτων για να δοκιμάσουμε κάποια σενάρια. Ο λόγος για τον οποίο δεν υλοποιήθηκε ο μηχανισμός ήταν εξαιτίας των περιορισμών που υπάρχουν στην πλατφόρμα XMT (δες υποκεφάλαιο 3.5.3). Συνεπώς αρκεστήκαμε στην λήψη μετρήσεων για διάφορα μεγέθη προβλημάτων. Ο κώδικας υλοποίησης φαίνεται στο Παράρτημα Γ. Επίσης για συντόμευση της διαδικασίας μεταγλώττισης και εκτέλεσης του κώδικα XMTC γίνεται χρήση ενός bash script ‘xmt_program.sh’ το οποίο φαίνεται στο Παράρτημα Γ. Οι μετρήσεις που πήραμε φαίνονται στον πίνακα που ακολουθεί (Πίνακας 6.1).

n	P(n)	clock cycles
4	4	874
8	8	1.130
16	16	1.327
32	32	1.599
64	64	2.010
128	128	2.360
256	256	3.849
512	512	4.700
1024	1024	7.328

Πίνακας 6.1 – Clock cycles προσομοιώσεων αλγορίθμου Χ



Γράφημα 6.1 – Γραφική αναπαράσταση των μετρήσεων Clock cycles (Πίνακας 6.1)

Ο λόγος για τον οποίο δεν έγινε σύγκριση με τα θεωρητικά πρότυπα είναι γιατί αντιμετωπίσαμε πρόβλημα στην δημιουργία του μηχανισμού πρόκλησης σφαλμάτων (εξαιτίας των περιορισμών της πλατφόρμας XMT) και γιατί είναι δύσκολη η εξαγωγή πληροφορίας σχετικά για τις μονάδες κόστους του κάθε επεξεργαστή ξεχωριστά.

Κεφάλαιο 7

Συμπεράσματα

7.1 Τελικά Συμπεράσματα

Στην παρούσα Ατομική Διπλωματική εργασία έχουμε υλοποιήσει και προσομοιώσει παράλληλους αλγορίθμους στην πλατφόρμα XMT. Χαρακτηριστικά στο Κεφάλαιο 4 έχουμε καταλήξει στο συμπέρασμα ότι ο τρόπος διαχείρισης της μνήμης επηρεάζει τον χρόνο ολοκλήρωσης του αλγορίθμου που προσομοιώνουμε. Συγκεκριμένα η έξυπνη διαχείριση της μνήμης, σε ποσότητα και τοπικότητα προσβάσεων αποτελούν παράμετροι που επηρεάζουν τον χρόνο ολοκλήρωσης.

Επίσης με την υλοποίηση του αλγορίθμου W στην πλατφόρμα XMT παρατηρήσαμε ότι ο χρόνος εκτέλεσης ασυμπτωτικά συνάδει με τα θεωρητικά πρότυπα. Επίσης με την δημιουργία του μηχανισμού πρόκλησης σφαλμάτων πετύχαμε την μελέτη συγκεκριμένων σεναρίων (υπερεκτίμησης, υποεκτίμησης) με σφάλματα. Με την προσομοίωση των σεναρίων αυτών, καταλήξαμε στο συμπέρασμα ότι σε σενάρια με υποεκτίμηση ο χρόνος ολοκλήρωσης του αλγορίθμου είναι ελαφρώς μεγαλύτερος σε σύγκριση με τα σενάρια υπερεκτίμησης. Επίσης παρατηρήσαμε ότι το κόστος και για τα δύο σενάρια που μελετήσαμε ήταν στα ίδια επίπεδα εξαιτίας του όμοιου αριθμού σφαλμάτων σε κάθε βήμα του αλγορίθμου.

Σχετικά με τον αλγόριθμό X πετύχαμε την υλοποίηση του στην πλατφόρμα XMT, όμως λόγω των περιορισμών της πλατφόρμας δεν καταφέραμε την υλοποίηση ενός

μηχανισμού πρόκλησης σφαλμάτων για την αξιολόγηση του. Πετύχαμε όμως την λήψη μετρήσεων για τον χρόνο ολοκλήρωσης του, και παρατηρήσαμε ότι συνάδει με τα θεωρητικά πρότυπα.

7.2 Προβλήματα και Πώς Αντιμετωπίστηκαν

Κατά την διάρκεια ολοκλήρωσης της παρών Ατομικής Διπλωματικής Εργασίας συνάντησα ποικίλα προβλήματα. Καταρχάς να τονίσουμε το γεγονός ότι η πλατφόρμα XMT βρίσκεται ακόμη υπό ανάπτυξη, άρα είναι λογικό να υπάρχουν ελλείψεις. Χαρακτηριστικό πρόβλημα ήταν ο περιορισμός που υπάρχει στο πλήθος των assembly εντολών που μπορούν να υπάρχουν σε ένα spawn-join block, και ήταν η αιτία για την μη υλοποίηση του μηχανισμού πρόκλησης σφαλμάτων. Επίσης το ότι δεν υποστηρίζονται βιβλιοθήκες (για παράδειγμα `<math.h>`) προκαλεί πρόβλημα στην συγγραφή του κώδικα. Χαρακτηριστικά στην υλοποίηση του αλγόριθμου X έπρεπε να υπολογίσω δυνάμεις του 2 πράγμα το οποίο πέτυχα με binary shifting. Τελειώνοντας θα ήθελα να τονίσω το ότι μελετήσαμε παραδείγματα τα οποία χρησιμοποιούν μέχρι 1024 επεξεργαστές, λόγο του περιορισμού στον προσομοιωτή. Εάν υπήρχε η δυνατότητα μελέτης μεγαλύτερων προβλημάτων θα είχαμε μια πιο ρεαλιστική άποψη για την συμπεριφορά της πλατφόρμας XMT.

7.3 Οφέλη Ατομικής Διπλωματικής Εργασίας

Με την ολοκλήρωση της παρούσας Ατομικής Διπλωματικής Εργασίας θα ήθελα να τονίσω τα οφέλη που αποκόμισα. Καταρχάς μου δόθηκε για πρώτη φορά η ευκαιρία για συγγραφή ατομικής διπλωματικής εργασίας. Στην πορεία έμαθα για τους κανόνες που διέπουν την συγγραφή μιας Ατομικής Διπλωματικής Εργασίας, για τον ορθό τρόπο λήψης πολλαπλών μετρήσεων και γενικότερα την μεθοδολογία που ακολουθείται στην ερευνητική μελέτη.

Ακόμη ασχολήθηκα με την πλατφόρμα XMT, η οποία υπάρχει υλοποιημένη σε πρωτότυπο FPGA, μέσω της οποίας έστω και με την χρήση του προσομοιωτή· μου έδωσε την δυνατότητα πρακτικής ενασχόλησης με τους παράλληλους αλγορίθμους. Ο παράλληλος τρόπος σκέψης που ανέπτυξα για την κατανόηση και υλοποίηση των αλγορίθμων X και W, άλλαξαν τον τρόπο με τον οποίο προσεγγίζω την επίλυση προβλημάτων. Το γεγονός ότι ο παραλληλισμός αποτελεί το μέλλον της πληροφορικής,

σίγουρα ο παράλληλος τρόπος σκέψης που ανέπτυξα θα με βοηθήσει στην μετέπειτα σταδιοδρομία μου.

7.4 Μελλοντική Εργασία

Πιθανόν στο μέλλον, όταν η πλατφόρμα XMT αναβαθμιστεί περαιτέρω, ίσως να είναι δυνατή η μελέτη των αλγορίθμων X και W σε μεγαλύτερου μεγέθους προβλήματα. Επίσης εάν μελλοντικά υποστηρίζονται συναρτήσεις ορισμένες από τον χρήστη εντός των spawn-join blocks ίσως είναι δυνατή η μελέτη σεναρίων που εκτελείται ‘δουλειά’ στα φύλλα. Εάν είναι δυνατή η μελέτη τέτοιων σεναρίων θα δύναται να εξαχθούν πιο ρεαλιστικά αποτελέσματα και συμπεράσματα.

Ίσως μια άλλη κατεύθυνση εργασίας είναι η χρήση των αποτελεσμάτων της παρούσας Ατομικής Διπλωματικής Εργασίας για σύγκριση με κάποια άλλη πλατφόρμα παράλληλου προγραμματισμού.

Μία ακόμη πρόταση για μελλοντική εργασία είναι η μελέτη σε περιβάλλον με συνεταιριστική επίλυση προβλημάτων. Πιο συγκεκριμένα, πολλαπλά υπολογιστικά συστήματα (τα οποία έχουν εγκατεστημένη την πλατφόρμα XMT) αναλαμβάνουν το κάθε ένα ξεχωριστά να ολοκληρώσει ένα μέρος της συνολικής εργασίας. Με τον τρόπο αυτό είναι δυνατή η αξιολόγηση της συμπεριφοράς ενός συνόλου υπολογιστών που αξιοποιεί την πλατφόρμα XMT.

Βιβλιογραφία

[1] Chryssis Georgiou and Alexander A. Shvartsman,.: Morgan & Claypool Publishers, 2010, ch. Chapter 4.

[2] XMT Research Team. Explicit Multi-Threading (XMT): A PRAM-On-Chip Vision - A Desktop Supercomputer. [Online]. http://www.umiacs.umd.edu/~vishkin/XMT/index.shtml

[3] Wikipedia. [Online]. http://en.wikipedia.org/wiki/Parallel_computing

[4] Chryssis Georgiou, Σημειώσεις μαθήματος EPL431.

[5] Wikipedia. [Online]. http://en.wikipedia.org/wiki/Flynn's_taxonomy

[6] Wikipedia. [Online]. http://en.wikipedia.org/wiki/Parallel_Random_Access_Machin

[7] Wikipedia. [Online]. http://en.wikipedia.org/wiki/Symmetric_multiprocessing#SMP-capable_processors

[8] Uzi Vishkin. Uzi Vishkin homepage. [Online]. http://www.umiacs.umd.edu/~vishkin/index.shtml#top

[9] Wikipedia. [Online]. http://en.wikipedia.org/wiki/XMTC

[10] Uzi Vishkin, George C. Caragea, and Bryant C. Lee, "Models for advancing PRAM and other algorithms into parallel programs for a PRAM-On-Chip platform," no. UMIACS-UMIACS-TR-2006-21, 2006.

[11] Uzi Vishkin, "Using Simple Abstraction to Reinvent Computing for Parallelism," *Communications of the ACM*, vol. 54, no. 1, January 2011.

[12] X. Wen and Uzi Vishkin, "FPGA-based prototype of a PRAM-on-chip processor," *Proceedings of the Fifth ACM Conference on Computing Frontiers (Ischia, Italy, May 5-7)*, pp. 55-66, 2008.

[13] A. Tzannes, G. Caragea, and Uzi Vishkin, "Lazy binary splitting: A run-time adaptive dynamic works-stealing scheduler.," *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming (Bangalore, India, jan. 9-14)*, pp. 179-189, 2010.

[14] G. Caragea, A. Tzannes, F. Keceli, R. Barua, and Uzi Vishkin, "Resource-aware compiler prefetching for many-cores," *Proceedings of the Ninth International Symposium on Parallel and Distributed Computing (Istanbul, Turkey, July 7-9)*, pp. 133-140, 2010.

[15] Aydin O. Balkan, Uzi Vishkin, George C. Caragea, and Alexandros Tzannes. XMT Toolchain Manual for XMTC Language, XMTC Compiler, XMT Simulator and Paraleap XMT FPGA Computer (Part 1 of 2) v.0.82.1 - October 27th, 2010. pdf manual. [Online]. http://www.umiacs.umd.edu/~vishkin/XMT/index.shtml

[16] Aydin O. Balkan, Uzi Vishkin, George C. Caragea, and Alexandros Tzannes. Programmer's Manual for XMTC Language, XMTC Compiler and Paraleap XMT FPGA Computer (Part 2 of 2) v.0.82.1 - October 27th, 2010. pdf manual. [Online]. http://www.umiacs.umd.edu/~vishkin/XMT/index.shtml

Παράρτημα Α

Σε αυτό το παράρτημα υπάρχουν οι XMTC κώδικες για τις υλοποιήσεις των εκδοχών του αλγορίθμου παράλληλης άθροισης, όπως αυτοί περιγράφονται στο Κεφάλαιο 4.

A.1 Αλγ. Παράλληλης Άθροισης parSum-A

```
#include <xmtc.h>

#define _n_512 0
//#define _print_mem 0

#if defined _n_4
    #define n 4
    #define log_n 2

#elif defined _n_8
    #define n 8
    #define log_n 3

#elif defined _n_16
    #define n 16
    #define log_n 4

#elif defined _n_32
    #define n 32
    #define log_n 5

#elif defined _n_64
    #define n 64
    #define log_n 6

#elif defined _n_128
    #define n 128
    #define log_n 7

#elif defined _n_256
    #define n 256
    #define log_n 8

#elif defined _n_512
    #define n 512
    #define log_n 9

#elif defined _n_1024
    #define n 1024
    #define log_n 10

#elif defined _n_2048
    #define n 2048
    #define log_n 11

#elif defined _n_4096
    #define n 4096
    #define log_n 12

#elif defined _n_8192
    #define n 8192
    #define log_n 13

#elif defined _n_16384
    #define n 16384
```

```
    #define log_n 14

#elif defined _n_32768
    #define n 32768
    #define log_n 15

#elif defined _n_65536
    #define n 65536
    #define log_n 16

#elif defined _n_131072
    #define n 131072
    #define log_n 17

#endif

int main(){

    int p, c, r, i;
    int num_rows, num_col;
    int sum[log_n + 1][n + 1]; // table size = (log_n + 1) * (n + 1)

    num_rows = log_n + 1;
    num_col = n + 1;
    p=n/2;

    //// initialization of array "sum"
    for (r=1; r<(num_rows); r++){
        spawn(0,n){
            sum[r][$]=0;
        }
    }
    spawn(0,n){
        if ($ == 0)
            sum[0][$]=0;
        else
            sum[0][$]=1;
    }
    //sum[0][1]=0;

    //appear on screen initial values of the variables
    #ifdef _print_mem
    printf("\nn = %d", n);
    printf("\nlog_n = %d", log_n);
    printf("\np = %d", p);
    printf("\nsum[%d][%d] : (log_n + 1) * (n + 1)\n", num_rows,
num_col);
    #endif

    //appear on screen contents of array "sum" before parallel section
    #ifdef _print_mem
    printf("\narray \"sum\" BEFORE parallel section:");
    printf("\n^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^\n");
    for (r=0; r<num_rows; r++){
        printf("row %d :", r);
        for (c=0; c < num_col; c++){
            printf("\t%d", sum[r][c]);
        }
        printf("\n");
    }
```

```
    #endif

    //################################
    //### parallel section BEGINS ###
    //################################
    r = 1;
    for (i=p; i>=1; i=i/2){

        spawn(1, i){
            int a, b;
            a = sum[r - 1][2 * $ - 1]; //read value located on 2*PID-1
            b = sum[r - 1][2 * $]; //read value located on 2*PID
            sum[r][$] = a + b; //write sum on location PID
        }////end of spawn

        r++;
    }///end of for loop
    //################################
    //###  parallel section ENDS  ###
    //################################

    //appear on screen contents of array "sum" after parallel section
    #ifdef _print_mem
    printf("\narray \"sum\" AFTER parallel section:");
    printf("\n^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^\n");
    for (r=0; r<num_rows; r++){
        printf("row %d :", r);
        for (c=0; c < num_col; c++){
            printf("\t%d", sum[r][c]);
        }
        printf("\n");
    }
    printf("\n");
    #endif
}
```

A.2 Αλγ. Παράλληλης Άθροισης parSum-B

```
#include <xmtc.h>

#define _n_512 0
//#define _print_mem 0

#if defined _n_4
    #define n 4
    #define log_n 2

#elif defined _n_8
    #define n 8
    #define log_n 3

#elif defined _n_16
    #define n 16
    #define log_n 4

#elif defined _n_32
    #define n 32
    #define log_n 5

#elif defined _n_64
    #define n 64
    #define log_n 6

#elif defined _n_128
    #define n 128
    #define log_n 7

#elif defined _n_256
    #define n 256
    #define log_n 8

#elif defined _n_512
    #define n 512
    #define log_n 9

#elif defined _n_1024
    #define n 1024
    #define log_n 10

#elif defined _n_2048
    #define n 2048
    #define log_n 11

#elif defined _n_4096
    #define n 4096
    #define log_n 12

#elif defined _n_8192
    #define n 8192
    #define log_n 13

#elif defined _n_16384
    #define n 16384
```

```
    #define log_n 14

#elif defined _n_32768
    #define n 32768
    #define log_n 15

#elif defined _n_65536
    #define n 65536
    #define log_n 16

#elif defined _n_131072
    #define n 131072
    #define log_n 17

#endif

int main(){

    int p, c, i;
    int sum_max_size, sum_last_index;
    int sum[1 << (log_n + 1)]; // table (log_n + 1)

    sum_max_size = 1 << (log_n + 1);
    sum_last_index = sum_max_size - 1;
    p=n/2;

    //// initialization of array "sum"
    spawn(0,sum_last_index){
        sum[$]=0;
    }
    //set values to be summed, so total_sum = n
    spawn(n,sum_last_index){
        sum[$] = 1;
    }
    /*/set values to be summed, so total_sum = 1+2+...+n
    spawn(n,sum_last_index){
        sum[$] = $ - (n - 1);
    }*/

    //appear on screen initial values of the variables
    #ifdef _print_mem
    printf("\nn = %d", n);
    printf("\nlog_n = %d", log_n);
    printf("\np = %d", p);
    printf("\nsum[%d] : 2^(log_n + 1)\n", sum_max_size);
    #endif

    //appear on screen contents of array "sum" before parallel section
    #ifdef _print_mem
    printf("\narray \"sum\" BEFORE parallel section:");
    printf("\n^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^\n");
    printf("row 0 :");
    for (c=0; c < sum_max_size; c++){
        printf("\t%d", sum[c]);
    }
    printf("\n");
    #endif

    //################################
    //### parallel section BEGINS ###
```

```
    //################################
    for (i=p; i>=1; i=i/2){

        spawn(1, i){
            int a, b, father_index;
            father_index = i + $ - 1;
            a = sum[2 * father_index];      //read left children value
            b = sum[2 * father_index + 1];  //read right children
value
            sum[father_index] = a + b;      //write sum on current
root location
        }////end of spawn

    }///end of for loop
    //################################
    //###  parallel section ENDS  ###
    //################################

    //appear on screen contents of array "sum" after parallel section
    #ifdef _print_mem
    printf("\narray \"sum\" AFTER parallel section:");
    printf("\n^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^\n");
    printf("row 0 :");
    for (c=0; c < sum_max_size; c++){
        printf("\t%d", sum[c]);
    }
    printf("\n\n");
    #endif
}
```

A.3 Αλγ. Παράλληλης Άθροισης parSum-C

```
#include <xmtc.h>

#define _n_512 0
//#define _print_mem 0

#if defined _n_4
    #define n 4
    #define log_n 2

#elif defined _n_8
    #define n 8
    #define log_n 3

#elif defined _n_16
    #define n 16
    #define log_n 4

#elif defined _n_32
    #define n 32
    #define log_n 5

#elif defined _n_64
    #define n 64
    #define log_n 6

#elif defined _n_128
    #define n 128
    #define log_n 7

#elif defined _n_256
    #define n 256
    #define log_n 8

#elif defined _n_512
    #define n 512
    #define log_n 9

#elif defined _n_1024
    #define n 1024
    #define log_n 10

#elif defined _n_2048
    #define n 2048
    #define log_n 11

#elif defined _n_4096
    #define n 4096
    #define log_n 12

#elif defined _n_8192
    #define n 8192
    #define log_n 13

#elif defined _n_16384
    #define n 16384
```

```
    #define log_n 14

#elif defined _n_32768
    #define n 32768
    #define log_n 15

#elif defined _n_65536
    #define n 65536
    #define log_n 16

#elif defined _n_131072
    #define n 131072
    #define log_n 17

#endif

int main(){

    int p, c, i, h;
    int power_of_two, offset;
    int num_col;
    int sum[n + 1]; // table (n + 1)

    num_col = n + 1;
    p=n/2;

    //// initialization of array "sum"
    spawn(0,n){
        if ($ == 0)
            sum[$]=0;
        else
            sum[$]=1;
    }
    //sum[7]=0;

    //appear on screen initial values of the variables
    #ifdef _print_mem
    printf("\nn = %d", n);
    printf("\nlog_n = %d", log_n);
    printf("\np = %d", p);
    printf("\nsum[%d] : (n + 1)\n", num_col);
    #endif

    //appear on screen contents of array "sum" before parallel section
    #ifdef _print_mem
    printf("\narray \"sum\" BEFORE parallel section:");
    printf("\n^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^\n");
    printf("row 0 :");
    for (c=0; c < num_col; c++){
        printf("\t%d", sum[c]);
    }
    printf("\n");
    #endif

    //################################
    //### parallel section BEGINS ###
    //################################
    h = 1; // this variable is used to calculate 2^h
    offset = 1;
    for (i=p; i>=1; i=i/2){
```

```
        power_of_two = 1 << h; // power_of_two = 2^h

        spawn(1, i){
            int id, a, b;
            id = (($ - 1) * (power_of_two))   + 1; //assign id to each
processor/thread
            a = sum[id];
            b = sum[id + offset];
            sum[id] = a + b; //store/write sum based on processor id
        }////end of spawn

        h++;
        offset = offset * 2;
    }///end of for loop
    //################################
    //###  parallel section ENDS  ###
    //################################

    //appear on screen contents of array "sum" after parallel section
    #ifdef _print_mem
    printf("\narray \"sum\" AFTER parallel section:");
    printf("\n^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^\n");
    printf("row 0 :");
    for (c=0; c < num_col; c++){
        printf("\t%d", sum[c]);
    }
    printf("\n\n");
    #endif
}
```

Α.4 Αλγ. Παράλληλης Άθροισης parSum-D

```
#include <xmtc.h>

#define _n_512 0
//#define _print_mem 0

#if defined _n_4
    #define n 4
    #define log_n 2

#elif defined _n_8
    #define n 8
    #define log_n 3

#elif defined _n_16
    #define n 16
    #define log_n 4

#elif defined _n_32
    #define n 32
    #define log_n 5

#elif defined _n_64
    #define n 64
    #define log_n 6

#elif defined _n_128
    #define n 128
    #define log_n 7

#elif defined _n_256
    #define n 256
    #define log_n 8

#elif defined _n_512
    #define n 512
    #define log_n 9

#elif defined _n_1024
    #define n 1024
    #define log_n 10

#elif defined _n_2048
    #define n 2048
    #define log_n 11

#elif defined _n_4096
    #define n 4096
    #define log_n 12

#elif defined _n_8192
    #define n 8192
    #define log_n 13

#elif defined _n_16384
    #define n 16384
```

```
    #define log_n 14

#elif defined _n_32768
    #define n 32768
    #define log_n 15

#elif defined _n_65536
    #define n 65536
    #define log_n 16

#elif defined _n_131072
    #define n 131072
    #define log_n 17

#endif

int main(){

    int p, c, r, i;
    int num_rows, num_col;
    int read_row, write_row;
    int sum[2][n + 1]; // table size = 2 * (n + 1)

    num_rows = 2;
    num_col = n + 1;
    p=n/2;

    //// initialization of array "sum"
    spawn(0,n){
        sum[1][$]=0;
    }
    spawn(0,n){
        if ($ == 0)
            sum[0][$]=0;
        else
            sum[0][$]=1;
    }
    //sum[0][7]=0;

    //appear on screen initial values of the variables
    #ifdef _print_mem
    printf("\nn = %d", n);
    printf("\nlog_n = %d", log_n);
    printf("\np = %d", p);
    printf("\nsum[%d][%d] : 2 * (n + 1)\n", num_rows, num_col);
    #endif

    //appear on screen contents of array "sum" before parallel section
    #ifdef _print_mem
    printf("\narray \"sum\" BEFORE parallel section:");
    printf("\n^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^\n");
    for (r=0; r<num_rows; r++){
        printf("row %d :", r);
        for (c=0; c < num_col; c++){
            printf("\t%d", sum[r][c]);
        }
        printf("\n");
    }
    #endif
```

```
    //################################
    //### parallel section BEGINS ###
    //################################
    read_row = 0;
    write_row = 1;
    for (i=p; i>=1; i=i/2){

        spawn(1, i){
            int a, b;
            a = sum[read_row][2 * $ - 1];   //read value located on
2*PID - 1
            b = sum[read_row][2 * $];       //read value located on
2*PID
            sum[write_row][$] = a + b;      //write sum on location
PID
        }////end of spawn

        // reverse read and write rows
        if (read_row == 0){
            read_row = 1;
            write_row = 0;
        }else{
            read_row = 0;
            write_row = 1;
        }
    }///end of for loop
    //################################
    //###  parallel section ENDS  ###
    //################################

    //appear on screen contents of array "sum" after parallel section
    #ifdef _print_mem
    printf("\narray \"sum\" AFTER parallel section:");
    printf("\n^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^\n");
    for (r=0; r<num_rows; r++){
        printf("row %d :", r);
        for (c=0; c < num_col; c++){
            printf("\t%d", sum[r][c]);
        }
        printf("\n");
    }
    printf("\n");
    #endif
}
```

A.5 Αλγ. Παράλληλης Άθροισης parSum-D.optimized

```
#include <xmtc.h>

#define _n_512 0
//#define _print_mem 0

#if defined _n_4
    #define n 4
    #define log_n 2

#elif defined _n_8
    #define n 8
    #define log_n 3

#elif defined _n_16
    #define n 16
    #define log_n 4

#elif defined _n_32
    #define n 32
    #define log_n 5

#elif defined _n_64
    #define n 64
    #define log_n 6

#elif defined _n_128
    #define n 128
    #define log_n 7

#elif defined _n_256
    #define n 256
    #define log_n 8

#elif defined _n_512
    #define n 512
    #define log_n 9

#elif defined _n_1024
    #define n 1024
    #define log_n 10

#elif defined _n_2048
    #define n 2048
    #define log_n 11

#elif defined _n_4096
    #define n 4096
    #define log_n 12

#elif defined _n_8192
    #define n 8192
    #define log_n 13

#elif defined _n_16384
    #define n 16384
```

```
    #define log_n 14

#elif defined _n_32768
    #define n 32768
    #define log_n 15

#elif defined _n_65536
    #define n 65536
    #define log_n 16

#elif defined _n_131072
    #define n 131072
    #define log_n 17

#endif

int main(){

    int p, r, c, i, h, n_div_p, values_population;
    int num_rows, num_col;
    int read_row, write_row;
    int sum[2][n + 1]; // table 2*(n + 1)

    num_rows = 2;
    num_col = n + 1;
    p= n / log_n;
    n_div_p = n / p;

    //// initialization of array "sum"
    spawn(0,n){
        if ($ == 0)
            sum[0][$]=0;
        else
            sum[0][$]=1;
    }
    spawn(0,n){
        sum[1][$]=0;
    }
    //sum[0][31]=3;
    //sum[0][32]=3;

        //appear on screen initial values of the variables
    #ifdef _print_mem
    printf("\nn = %d", n);
    printf("\nlog_n = %d", log_n);
    printf("\np = %d ,\tlogo seiriakou vimatos   p = n/log_n", p);
    printf("\nsum[2][%d] : 2*(n + 1)", num_col);
    printf("\nn_div_p = %d\n", n_div_p);
    #endif

    //appear on screen contents of array "sum" before parallel section
    #ifdef _print_mem
    printf("\narray \"sum\" BEFORE parallel section:");
    printf("\n^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^\n");
    for (r=0; r<num_rows; r++){
        printf("row %d :", r);
        for (c=0; c < num_col; c++){
            printf("\t%d", sum[r][c]);
        }
        printf("\n");
```

```
    }
    #endif


    //--------------------------------
    //---  seiriako vima BEGINS   ---
    //--------------------------------
    //step-01: p processors sum n/p values each
    spawn(1,p){
        int id, x, tmp_sum;
        tmp_sum=0;
        id = (($-1)*(n_div_p))+1;   //assign id to each processor (one
processor per n/p values)
        for ( x=id; x<id+(n_div_p); x++ ){  //each processor sums p
values - serial
            tmp_sum += sum[0][x];
        }
        sum[1][$] = tmp_sum;
    }
    //--------------------------------
    //---   seiriako vima ENDS    ---
    //--------------------------------



    //  ____________________________________
    // | endiameso-metavatiko vima BEGINS|
    // ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    // metafora ton timon pou vriskonte sto telos (kai den exun
a8ristei) stin
    // arxi, amesos meta ta p prota stoixeia (afto simvennei gia
paradeigma otan n=32)
    spawn(1,n-(p*n_div_p)){
        sum[1][p+$]=sum[0][n+1-$];
    }
    //  ____________________________________
    // | endiameso-metavatiko vima ENDS  |
    // ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^



    //################################
    //### parallel section BEGINS ###
    //################################
    read_row = 1;
    write_row = 0;
    values_population = (n-(p*n_div_p)) + p;
    i = values_population / 2;
    while (i>=1){

        spawn(1, i){
            int a, b;
            a = sum[read_row][2 * $ - 1];   //read value located on
2*PID - 1
            b = sum[read_row][2 * $];       //read value located on
2*PID
            sum[write_row][$] = a + b;      //write sum on location
PID
            if ( ($==1)&&((values_population % 2)!=0) ){
```

```
                sum[write_row][1] = sum[write_row][1] +
sum[read_row][values_population];
            }
        }////end of spawn

        // reverse read and write rows
        if (read_row == 0){
            read_row = 1;
            write_row = 0;
        }else{
            read_row = 0;
            write_row = 1;
        }

        values_population = i;
        i=i/2;
    }///end of "while" loop
    //################################
    //###  parallel section ENDS  ###
    //################################

    //appear on screen contents of array "sum" after parallel section
    #ifdef _print_mem
    printf("\narray \"sum\" AFTER parallel section:");
    printf("\n^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^\n");
    for (r=0; r<num_rows; r++){
        printf("row %d :", r);
        for (c=0; c < num_col; c++){
            printf("\t%d", sum[r][c]);
        }
        printf("\n");
    }
    printf("\n");
    #endif
}
```

Παράρτημα Β

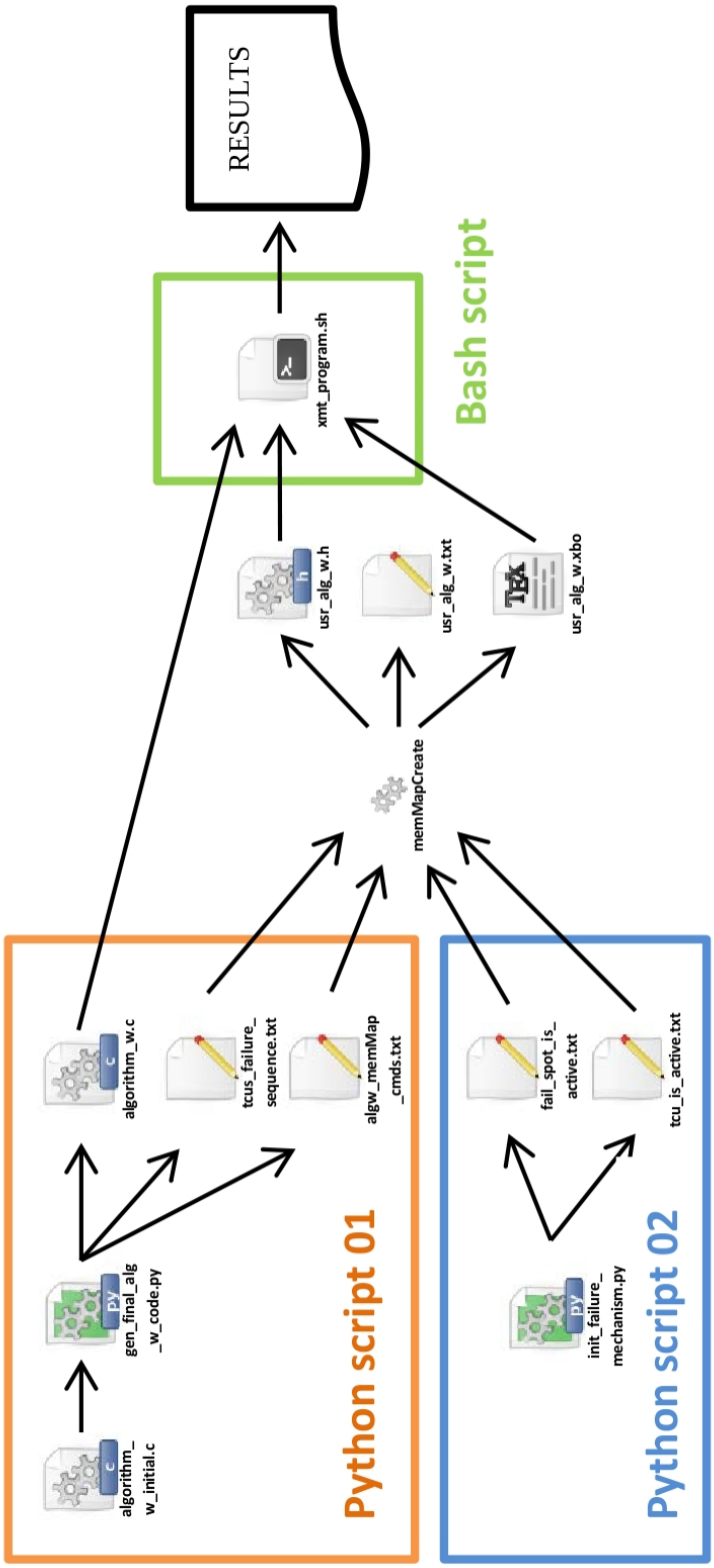
B.1 Οδηγός Εκτέλεσης Υλοποίησης Αλγορίθμου W

Προκειμένου να εκτελέσετε την υλοποίηση του αλγορίθμου W θα πρέπει να εκτελέσετε τις ακόλουθες εντολές:

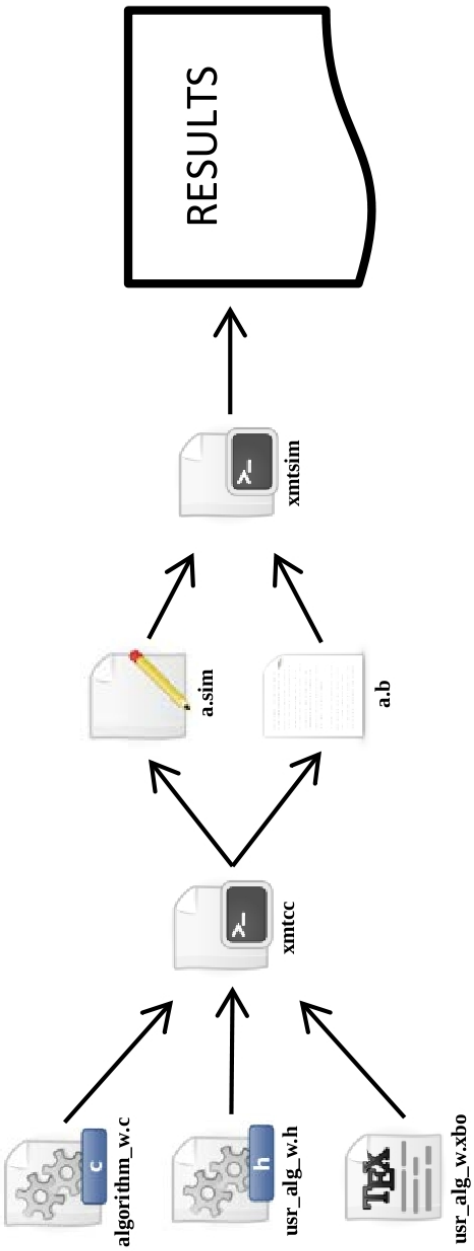
```
Terminal$ ./gen_final_alg_w_code.py [hnfmp]

Terminal$ ./init_failure_mechanism.py [hn]

Terminal$ memMapCreate < algw_memMap_cmds.txt

Terminal$ ./xmt_program.sh algorithm_w
```

Για περισσότερες πληροφορίες σχετικά με τις εντολές .py ανατρέξτε στα εισαγωγικά σχόλια εντός του αντίστοιχου python αρχείου. Δηλαδή για την εντολή `./gen_final_alg_w_code.py` ανατρέξτε στα εισαγωγικά σχόλια εντός του αρχείου gen_final_alg_w_code.py για πληροφορίες σχετικά με τις παραμέτρους. Παρομοίως και για την εντολή `./init_failure_mechanism.py`. Αυτές οι πληροφορίες εμφανίζονται και στα παραρτήματα B.4 και B.5.

B.2 Ροή Δεδομένων για Υλοποίηση Αλγορίθμου W



B.3 Ροή Δεδομένων για “xmt_program.sh”



B.4 Python Script “gen_final_alg_w_code.py”

```
#!/usr/bin/python -u

#    ____________________________________________________________
#   {_____   |##########################################|   _____}
#         \  |###                                    ###|  /
#       __/  |### Name : Efstathios Ioakim           ###|  \__
#      {__   |### Year : 2012                        ###|   __}
#         \  |###                                    ###|  /
#    _____/  |###                                    ###|  \_____
#   {________|##########################################|________}
#            `````````````````````````````````````````````

#
# PROGRAMMER COMMENTS:
#
# Script Main Task:
# """""""""""""""""""
# This PYTHON (.py) script is used to generate a file containing the final
xmtc
# code (algorithm w) and also generates two more files (used by memMap tool)
#
# Output of this script are three files (one .c file and two .txt files):
# 1) file "algorithm_w.c" contains the final xmtc code (algorithm w) that will
#    be compiled and simulated later.
#
# 2) file "tcus_failure_sequence.txt" contains a random generated failure
#    sequence. This sequence is used by failure mechanism. For example the
file
#    might look something like the following string (suppose n=8)
#    "0 4 3 6 5 1 7 2 8". In case you notice the first value of the string is
#    always zero "0" cause the (.txt) file represents the values that will be
#    used to initalize the array "tcus_failure_sequence[n+1]".
#
# 3) file "algw_memMap_cmds.txt" contains the keystrokes for generating the
#    particular set of memory files (header and binary files) using the
#    'memMapCreate' program. This text file will be fed into the
'memMapCreate'
#    program using basic redirection operators.
#
# IMPORTANT: instead of using the term 'Processors' i prefer using the term
# """""""""" 'tcus' (tread control units) in order to be closer to xmtc
platform
#            terminology.
#
# USAGE:
# """""""
# Terminal$ ./gen_final_alg_w_code.py [hnf]
#
#   -n #   : set maximun Thread Control Units number (default n=8). MUST be
#            power of 2
#
#   -f #   : set maximum number of failures per fail spot
#
#   -h     : display powers of 2
#
#   -m     : failure mechanism working mode (accepted values '1' or '2')
#
#   -p     : paronomastis gia tin dieresi otan to failure mechanism eine sto
mode '2'
#

###############################################################################
```

```
### M O D U L E S ##############################################################
################################################################################
import os
import re
import sys
import stat
import getopt
import math
import random
from operator import itemgetter, attrgetter

################################################################################
### G L O B A L   V A R I A B L E S ############################################
################################################################################
# Maximum number of fail spots inside algorith w code
GV_fail_spots_num = 29

# Filename where the initial code of algorithm w is located to
GV_alg_w_source_filename = "algorithm_w_initial.c"

# Filename where the final code of algorithm w will be stored to (this code is
# the result of processing the initial algorithm w code)
GV_alg_w_output_filename = "algorithm_w.c"

# Filename where the keystrokes will be stored to (fed later to 'memMapCreate'
# program to generate memory files)
GV_alg_w_memMap_cmds_filename = "algw_memMap_cmds.txt"

# Filename where the random tcu fail sequence will be stored to
GV_tcus_fail_seq_output_filename = "tcus_failure_sequence.txt"

################################################################################
### U S E R   D E F I N E D   F U N C T I O N S ################################
################################################################################
# Fuction that appears onscreen (terminal) the powers of 2 (via -h parameter,
# check 'usage' section under 'programmers comments'at top of this script)
def display_powers():
    for tmp_i in range(1,16):
        print "n=" + str(int(math.pow(2,tmp_i))) + " log_n=" + str(tmp_i)

################################################################################
### M A I N   B O D Y ##########################################################
################################################################################
# >>> MAIN BODY BEGIN <<<

# String variables that will be put (replace some matchings) inside the
# algorithm w code
str_global_vars = """#define n {0}
#define log_n {1}
// <> end global variables"""
str_fail_spot = """
if ( (fail_spot_is_active[#]) && (fail_index < n) ){
    if ( fps_mode == 2 ){
        number_of_failures_per_spot = ((n - fail_index) + 1) /
fps_paronomastis;
        if (number_of_failures_per_spot == 0)
            number_of_failures_per_spot = 1;
    }
    for (tmp_count = 1; tmp_count <= number_of_failures_per_spot;
tmp_count++){
        if ( fail_index < n ){
            tcu_is_active[tcus_failure_sequence[fail_index]] = 0;
            fail_index++;
        }else{
            break;
        }
    }//end of "for" loop
```

```
}"""


# String variables that will be used in regex expressions
# Some of this will be used as matching strings and other as replace strings
str_if_tcu_active_begin_pattern = "//< if tcu active"
str_if_tcu_active_end_pattern = "//> endif tcu active"
str_if_tcu_active_begin_code = "if ( tcu_is_active[$] ){"
str_if_tcu_active_end_code = "}"

# List of tuples, used for random tcus fail sequence generation
fail_list = []

# If older generated algorithm w file exists, remove it in order to create a
new one
if os.path.exists(GV_alg_w_output_filename):
    os.remove(GV_alg_w_output_filename)

# Open output file
outp_file = open(GV_alg_w_output_filename,"w")

# Read initial algorithm w code
inp_file = open(GV_alg_w_source_filename,"r")
alg_w_code = inp_file.read()
inp_file.close()

# Get command line arguments
n = 8 #default value for 'n'
fails_per_spot = 0 #default value for 'failures per spot'
fps_mode = 1 #default value for 'fail mechanism mode'
fps_paronomastis = 1 #default value for 'fps_paronomastis'
flag_n = 0 #flag to detect user given value for 'n'
flag_f = 0 #flag to detect user given value for 'failures per spot'
flag_m = 0 #flag to detect user given value for 'fail mechanism mode'
flag_p = 0 #flag to detect user given value for 'fps_paronomastis'
opts, extraparams = getopt.getopt(sys.argv[1:], "hn:f:m:p:")
#print 'Opts:',opts
#print 'Extra parameters:',extraparams
print ""
for o, a in opts:
    if o == "-n":
        n = int(a)
        flag_n = 1
    elif o == "-h":
        display_powers()
        exit()
    elif o == "-f":
        fails_per_spot = int(a)
        flag_f = 1
    elif o == "-m":
        fps_mode = int(a)
        flag_m = 1
    elif o == "-p":
        fps_paronomastis = int(a)
        flag_p = 1

# Appear warning messages in order to inform user about automatic
initialization
# of failure mechanism's data structures
if flag_n == 0:
    print " Warning: option '-n' not given, value of 'n' set to default (n=8)"
if (flag_f == 0) and (fps_mode == 1):
    print " Warning: option '-f' not given, value of 'failures per spot' set
to default (f=0)"
if flag_m == 0:
```

```
    print " Warning: option '-m' not given, value of 'fail mechanism mode' set
to default (m=1)"
if (flag_p == 0) and (fps_mode == 2):
    print " Warning: option '-p' not given, value of 'fps_paronomastis' set to
default (p=1)"

# Insert global variables (n, log_n) to alg_w code
str_global_vars = str_global_vars.format(n, int(math.log(n,2)) )
tmp_result = re.subn("// <> end global variables", str_global_vars,
alg_w_code, 1)
alg_w_code = tmp_result[0]

# Insert code for "if tcu is active" using regex
tmp_result = re.subn(str_if_tcu_active_begin_pattern,
str_if_tcu_active_begin_code + str_if_tcu_active_begin_pattern, alg_w_code)
alg_w_code = tmp_result[0]
tmp_result = re.subn(str_if_tcu_active_end_pattern, str_if_tcu_active_end_code
+ str_if_tcu_active_end_pattern, alg_w_code)
alg_w_code = tmp_result[0]

# Generate random tcus fail sequence
for i in range(1, n+1):
    fail_list.append( (random.random(), int(i)) )

#print fail_list #debug
# Sort the list based on random item (equally to random sorting)
fail_list.sort(key=itemgetter(0))
#print fail_list #debug

###### the code below is still left hear in case i need it in future
#write tcu random fail sequence to alg_w code
#tmp_str_pre = "// <> end tcus failure sequence initialization"
#tmp_counter = 1
#for each_item in fail_list:
#   tmp_new_str = "tcus_failure_sequence[" + str(tmp_counter) + "] = " +
str(each_item[1]) + ";" + """
#   // <> end tcus failure sequence initialization"""
#   tmp_result = re.subn(tmp_str_pre, tmp_new_str, alg_w_code, 1)
#   alg_w_code = tmp_result[0]
#   tmp_counter += 1

# Write tcu random fail sequence to 'tcus_failure_sequence.txt'
# Open output file
outp_tcu_fail_seq_file = open(GV_tcus_fail_seq_output_filename,"w")

# Generate string
tmp_new_str = "0"
for each_item in fail_list:
    tmp_new_str += " " + str(each_item[1])

# Write string to output file
outp_tcu_fail_seq_file.write(tmp_new_str)
# Close output file
outp_tcu_fail_seq_file.close()
# Set persmissions for output file
os.chmod(GV_tcus_fail_seq_output_filename,
stat.S_IRWXU|stat.S_IRWXG|stat.S_IRWXO)



# Insert fail spots specific code (via the use of regex)
for i in range(1, GV_fail_spots_num + 1):
    tmp_str_pre = "/// <> fail spot " + str(i)

    tmp_r1 = re.subn("#", str(i), str_fail_spot)
    tmp_str_after = tmp_r1[0]
```

```
    tmp_new_str = tmp_str_pre + tmp_str_after

    tmp_result = re.subn(tmp_str_pre, tmp_new_str, alg_w_code, 1)
    alg_w_code = tmp_result[0]

# Write final code for alg w to output file
outp_file.write(alg_w_code)

# Close output file
outp_file.close()

# Set persmissions for output file
os.chmod(GV_alg_w_output_filename, stat.S_IRWXU|stat.S_IRWXG|stat.S_IRWXO)



# Create a file that contains the keystrokes for generating the particular set
# of memory files (header and binary files) using the 'memMapCreate' program.

# Open output file
mem_create_file = open(GV_alg_w_memMap_cmds_filename, "w")

tmp_result = """1
4
usr_alg_w
<
2
2
task
1
{0}
0
y
<
2
2
count
1
{1}
0
y
<
2
2
done
1
{2}
0
y
<
2
2
aux
1
{3}
0
y
<
2
2
pnum
1
{4}
0
y
<
2
```

```
2
k
1
{5}
0
y
<
2
2
i
1
{6}
0
y
<
2
2
i1
1
{7}
0
y
<
2
2
i2
1
{8}
0
y
<
2
2
subt
1
{9}
0
y
<
2
1
size
y
0
<
2
1
max_tcu_num
y
{10}
<
2
1
depth
y
0
<
2
2
tcu_is_active
1
{11}
tcu_is_active.txt
y
<
2
2
```

```
fail_spot_is_active
1
{12}
fail_spot_is_active.txt
y
<
2
1
number_of_failures_per_spot
y
{13}
<
2
2
tcus_failure_sequence
1
{14}
tcus_failure_sequence.txt
y
<
2
1
tmp_count
y
0
<
2
1
fail_index
y
1
<
2
1
fps_mode
y
{15}
<
2
1
fps_paronomastis
y
{16}
<
2
H
<
2
B
<
q
"""

# Insert variables values
tmp_result = tmp_result.format(n+1, 2*n, 2*n, 2*n, n+1, n+1, n+1, n+1, n+1,
n+1, n, n+1, GV_fail_spots_num +1, fails_per_spot, n+1, fps_mode,
fps_paronomastis)

# Write string to file
mem_create_file.write(tmp_result)

#set persmissions for file
os.chmod(GV_alg_w_memMap_cmds_filename,
stat.S_IRWXU|stat.S_IRWXG|stat.S_IRWXO)

# put some new lines in stdout
print "\n\n"
```

```
# >>> MAIN BODY END <<<
```

B.5 Python Script “init_failure_mechanism.py”

```
#!/usr/bin/python -u

#    _____________________________________________________________
#   {_____   |##########################################|   _____}
#         \  |###                                    ###|  /
#       __/  |### Name : Efstathios Ioakim            ###|  \__
#      {__   |### Year : 2012                         ###|   __}
#         \  |###                                    ###|  /
#    _____/  |###                                    ###|  \_____
#   {________|##########################################|________}
#            `````````````````````````````````````````````

#
# PROGRAMMER COMMENTS:
#
# Script Main Task:
# """""""""""""""""""
# This PYTHON (.py) script is used to initialize the failure mechanism data
# structures.
#
# Output of this script are two (.txt) files:
# 1) file "tcu_is_active.txt" contains the active status of each tcu uppon the
#    begining of execution of xmtc algorithm-program. For example if all tcus
#    are active the file will contain the following string (suppose n=8)
#    "0 1 1 1 1 1 1 1 1 ". If only tcu #1 is active then it will contain the
#    following string (suppose n=8) "0 1 0 0 0 0 0 0 0 ". In case you notice
#    the first value of the string is always zero "0" cause the (.txt) file
#    represents the values that will be used to initalize the array
#    "tcu_is_active[n+1]".
#
# 2) file "fail_spot_is_active.txt" contains the active status of each failure
#    spot located inside the xmtc algorithm-program. For example if all fail
#    spots are active the file will contain the following string
#    "0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ". If only
#    fail spot #1 is active then it will contain the following string
#    "0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ". In case
you
#    notice the first value of the string is always zero "0" cause the (.txt)
#    file represents the values that will be used to initalize the array
#    "fail_spot_is_active[fail_spot_num + 1]", where 'fail_spot_num = 29'.
#
# IMPORTANT: instead of using the term 'Processors' i prefer using the term
# """""""""" 'tcus' (tread control units) in order to be closer to xmtc
platform
#            terminology.
#
# USAGE:
# """""""
# Terminal$ ./init_failure_mechanism.py [hn]
#
#   -n #   : set maximun Thread Control Units number (default n=8). MUST be
#            power of 2
#
#   -h     : display powers of 2
#

#############################################################################
### M O D U L E S ###########################################################
#############################################################################
import os
import re
import sys
```

```
import stat
import getopt
import math
import random
from operator import itemgetter, attrgetter

################################################################################
### G L O B A L   V A R I A B L E S ############################################
################################################################################
# Maximum number of fail spots inside algorith w code
GV_fail_spot_num = 29

# Filename where the values (active status for each tcu) given by user will be
# stored to.
GV_tcu_is_active_filename = "tcu_is_active.txt"

# Filename where the values (active status for fail spot) given by user will
be
# stored to.
GV_fail_spot_is_active_filename = "fail_spot_is_active.txt"

################################################################################
### U S E R   D E F I N E D   F U N C T I O N S ################################
################################################################################
# Fuction that appears onscreen (terminal) the powers of 2 (via -h parameter,
# check 'usage' section under 'programmers comments'at top of this script)
def display_powers():
    for tmp_i in range(1,16):
        print "n=" + str(int(math.pow(2,tmp_i))) + " log_n=" + str(tmp_i)

# Function that appears onscreen (terminal) a menu
def print_menu():
    print "\n failure mechanish menu (w algorithm)"
    print " ================================="
    print " [1] - edit fail spot's flags (num of fail spots =
"+str(GV_fail_spot_num)+")"
    print " [2] - edit tcus active status (num of tcus = n)"
    print " [3] - exit"

################################################################################
### M A I N   B O D Y ##########################################################
################################################################################
# >>> MAIN BODY BEGIN <<<

# Define lists to store given values by user
fail_spot_is_active_list = []
tcu_is_active_list = []

# Get command line arguments
n = 8 #default value for 'n'
flag = 0 #flag to detect user given argument
opts, extraparams = getopt.getopt(sys.argv[1:], "hn:")
#print 'Opts:',opts
#print 'Extra parameters:',extraparams
print ""
for o, a in opts:
    if o == "-n":
        n = int(a)
        flag = 1
    elif o == "-h":
        display_powers()
        exit()

# Appear warning messages in order to inform user about automatic
initialization
# of failure mechanism's data structures
if flag == 0:
```

```
    print " Warning: option '-n' not given, value of 'n' set to default (n=8)"
print " Warning: By default all fail spots are deactivated"
print " Warning: By default all tcus are active"

# Automatic initialization of failure mechanism's data structures (lists)
for i in range(0, GV_fail_spot_num + 1):
    fail_spot_is_active_list.append( (i, 0) )
tcu_is_active_list.append( (0, 0) )
for i in range(1, n + 1):
    tcu_is_active_list.append( (i, 1) )

# Infite loop that handles the main menu of this script
# User can exit this loop only if selects 'exit' option from menu (give number
# of this option)
usr_choise = 0
while ( 1 == 1 ):
    # Draw menu onscreen (terminal)
    print_menu()

    # Read user's choise
    usr_choise = int(raw_input("\n > "))

    # The following part of "if elif...else" code has the responsibility of
    # of activating the funtion that user selected from main menu
    if (usr_choise == 1):
        # In this case user selected 'edit fail spot's flags' option from menu
        # Read user desired range to change flags
        range1 = int(raw_input("fail spot # (begin) : "))
        range2 = int(raw_input("fail spot # (end)   : "))
        # Check if range parameters given by user (left and right limits) are
        # between allowed values
        if ( (range1 >= 1) and (range2 <= GV_fail_spot_num) ):
            # Get flag value given by user
            flag_value = int(raw_input("flag value (0/1) : "))
            # Change value of each flag in given range
            for i in range (range1, range2 + 1):
                #print "num="+str(i)
                fail_spot_is_active_list[i] = (i, flag_value)
    elif (usr_choise == 2):
        # In this case user selected 'edit tcus active status' option from
menu
        # Read user desired range to change flags
        range1 = int(raw_input("tcu # (begin) : "))
        range2 = int(raw_input("tcu # (end)   : "))
        # Check if range parameters given by user (left and right limits) are
        # between allowed values
        if ( (range1 >= 1) and (range2 <= n) ):
            # Get tcu flag value given by user
            tcu_flag = int(raw_input("tcu active status (0/1) : "))
            # Change value of each tcu flag in given range
            for i in range (range1, range2 + 1):
                tcu_is_active_list[i] = (i, tcu_flag)
    elif (usr_choise == 3):
        # In this case user selected 'exit' option from menu, so we break the
        # infinite loop.
        break
    else:
        # In case no match was found from previous checks, this means user
        # entered invalid option, so i appear error message
        print " Please choose a valid option ! ! !"



# Write tcu active flags to 'fail_spot_is_active.txt'

# Open output file
outp_fail_spot_is_active_file = open(GV_fail_spot_is_active_filename,"w")
```

```
tmp_new_str = ""
for each_item in fail_spot_is_active_list:
    tmp_new_str += str(each_item[1]) + " "
# Write string to output file
outp_fail_spot_is_active_file.write(tmp_new_str)
# Close output file
outp_fail_spot_is_active_file.close()
# Set persmissions for output file
os.chmod(GV_fail_spot_is_active_filename,
stat.S_IRWXU|stat.S_IRWXG|stat.S_IRWXO)



# Write tcu active flags to 'tcu_is_active.txt'

# Open output file
outp_tcu_is_active_file = open(GV_tcu_is_active_filename,"w")
tmp_new_str = ""
for each_item in tcu_is_active_list:
    tmp_new_str += str(each_item[1]) + " "
# Write string to output file
outp_tcu_is_active_file.write(tmp_new_str)
# Close output file
outp_tcu_is_active_file.close()
# Set persmissions for output file
os.chmod(GV_tcu_is_active_filename, stat.S_IRWXU|stat.S_IRWXG|stat.S_IRWXO)

# Appear a thank you message
print "\n Thank you for using init failure mechanism script ! ! !\n\n\n"

# >>> MAIN BODY END <<<
```

B.6 Bash Script “xmt_program.sh”

```
#!/bin/bash

#    _____________________________________________________
#   {_____   |##########################################|   _____}
#         \  |###                                    ###|  /
#       __/  |### Name : Efstathios Ioakim           ###|  \__
#      {__   |### Year : 2012                        ###|   __}
#         \  |###                                    ###|  /
#    _____/  |###                                    ###|  \_____
#   {________|##########################################|________}
#            `````````````````````````````````````````

#
# PROGRAMMER COMMENTS:
#
# Script Main Task:
# """""""""""""""""""
# This BASH(.sh) script is used for compiling and simulating xmtc code
#

# dump to file vars (xmtsim)
# -hexdump outp_mnimi -dumpvar tcu_is_active -dumpvar task

################################################################################
##
### SCRIPT BEGINS
#####################################################################
################################################################################
##
#clear contents onscreen (terminal)
clear

#set params flag to zero
check_flag=0

#check if parameter was given by user
if [ $1 = "algorithm_w" ]; then
    #set program name equal to parameter
    program_name="algorithm_w.c"
    #set parameter detect flag to TRUE
    check_flag=1
fi

#check if given parameter is what expected (check via flag)
# => if TRUE do compiling and simulation
# => if FALSE appear error message
if [ $check_flag = 1 ]; then
    #compile the xmtc code, including header file and binary memory map file
    xmtcc $program_name -include usr_alg_w.h usr_alg_w.xbo
    #begin simulation of program
    xmtsim -cycle -binload a.b a.sim
else
    #appear bad parameter error message on screen
    echo " Error : Bad parameter!"
fi
################################################################################
##
### SCRIPT ENDS
#####################################################################
##
```

B.7 Αρχικός Κώδικας XMTC Αλγορίθμου W (algorithm_w_initial.c)

```
#include <xmtc.h>

//    _____________________________________________________________
//   {_____   |#########################################|   _____}
//         \  |###                                   ###|  /
//       __/  |### Name : Efstathios Ioakim           ###|  \__
//      {__   |### Year : 2012                        ###|   __}
//         \  |###                                   ###|  /
//    _____/  |###                                   ###|  \_____
//   {________|#########################################|________}
//            `````````````````````````````````````````

//#############################################################################
##
//# G L O B A L    V A R I A B L E S
##########################################
//#############################################################################
##
#define fail_spot_num 29
//#define _print_mem 0
//#define _cost 0
// <> begin global variables
// <> end global variables

int main(){
    #ifdef _cost
    int sum;
    int sum_c;
    #endif

//#############################################################################
    //# shared integer array

//#############################################################################
/*  int task[n+1];  //task[1..n]
    int count[2*n]; //count[1..2n-1]
    int done[2*n];  //done[1..2n-1]
    int aux[2*n];   //aux[1..2n-1]
*/  //-------------------------------------------------------------------------
--


//#############################################################################
    //# private integer: each mem cell is used as private integer for each TCU

//#############################################################################
/*  int pnum[n+1];  //pnum[1..n]
    int k[n+1];     //k[1..n]
    int i[n+1];     //i[1..n]
    int i1[n+1];    //i1[1..n]
    int i2[n+1];    //i2[1..n]
    int subt[n+1];  //subt[1..n]
    int size;       //size
*/  //-------------------------------------------------------------------------
--


//#############################################################################
    //# miscellaneous variables

//#############################################################################
/*  int max_tcu_num; // maximum number of tcus (=n)
```

```
    int depth; // used to move through binary tree (phase w1 and w4
    int tcu_is_active[n+1]; // int flag used for TCUs failures mechanism
    int fail_spot_is_active[fail_spot_num + 1]; // int flag used to activate
fail spots
    int number_of_failures_per_spot; // number of tcu failures per fail spot
    int tcus_failure_sequence[n+1]; // int array containing the TCUs failure
sequence
    int tmp_count; // counter used in failure spots
    int fail_index; // index used to move through tcus_failure_sequence array
*/  //-------------------------------------------------------------------------
--


//############################################################################
    //# shared integer array initialization

//############################################################################
/*  spawn(0, n){
        task[$] = 0;    //task[n+1]
    }
    spawn(0, 2*n-1){
        count[$] = 0;   //count[2n]
        done[$] = 0;    //done[2n]
        aux[$] = 0;     //done[2n]
    }
*/  //-------------------------------------------------------------------------
--


//############################################################################
    //# miscellaneous variables initialization

//############################################################################
/*  max_tcu_num = n;
    depth = 0;
    spawn(0, n){ // all TCUs initially are enable
        tcu_is_active[$] = 1;
    }
    spawn(0, fail_spot_num){ // all fail spots are disable
        fail_spot_is_active[$] = 0;
    }
    number_of_failures_per_spot = 4;
    fail_spot_is_active[2] = 1; //enable failure spot 25 for debuging purposes

    // <> begin tcus failure sequence initialization
    tcus_failure_sequence[0] = 0;
    // <> end tcus failure sequence initialization

    fail_index = 1;
*/
    //-------------------------------------------------------------------------
--

    #ifdef _cost
    sum=0;
    sum_c=1;
    printf("sum=%d\n",sum);
    #endif

    // > > >   B E G I N   O F   M A I N   B O D Y   < < <

    /// <> fail spot 1


//############################################################################
    //# initial work item for PID
```

```
//#############################################################################
    spawn(1, max_tcu_num){
    //< if tcu active
        k[$] = $;   // k := PID; --inital work item for PID
    //> endif tcu active
    }
    //-------------------------------------------------------------------------
--

    /// <> fail spot 2


//#############################################################################
    //# W3

//#############################################################################
    spawn(1, max_tcu_num){
    //< if tcu active
        task[k[$]] = 1; // perform a task
    //> endif tcu active
    }
    //-------------------------------------------------------------------------
--

    /// <> fail spot 3


//#############################################################################
    //# W4:
    //# """
    //# private integers used:
    //#  >>> i, i1, i2 --parent and left/right child indices

//#############################################################################
    // mark the leaf done
    spawn(1, max_tcu_num){
    //< if tcu active
        i1[$] = k[$] + (n - 1); // leaf index
        done[i1[$]] = 1; // done!
    //> endif tcu active
    }

    /// <> fail spot 4

    // traverse the tree from leaf to root
    for (depth=1; depth <= log_n; depth++){

        /// <> fail spot 5

        spawn(1, max_tcu_num){
        //< if tcu active
            i[$] = i1[$] / 2;
            if ( (2*i[$]) == i1[$] ){ // came from ...
                i2[$] = i1[$] + 1; // ...left
            }else{
                i2[$] = i1[$] - 1; // ...right
            }

            // update progress tree and advance
            done[i[$]] = done[i1[$]] + done[i2[$]]; // update
            i1[$] = i[$]; // advance to parent
        //> endif tcu active
        }

        /// <> fail spot 6
```

```
    }
    //-------------------------------------------------------------------------
--

    /// <> fail spot 7

    #ifdef _cost
    sum=sum+(max_tcu_num - fail_index + 1);
    printf("%d alive=%d\tsum=%d\n", sum_c, max_tcu_num - fail_index + 1, sum);
    sum_c++;
    #endif


//#############################################################################
    //# Main Loop

//#############################################################################
    while ( done[1] != n ){ // Main loop

        /// <> fail spot 8


//#############################################################################
        //# W1:
        //# """
        //# private integers used:
        //#  >>> i, i1, i2 --parent and left/right child indices
        //#  >>> subt --running subtree total

//#############################################################################
        spawn(1, max_tcu_num){
        //< if tcu active
            i1[$] = $ + (n - 1); // start at leaf
            pnum[$] = 1; // assume this PID is number 1
            subt[$] = 1; // count PID once
        //> endif tcu active
        }

        /// <> fail spot 9

        // traverse the tree from leaf to root
        for (depth=1; depth <= log_n; depth++){

            /// <> fail spot 10

            spawn(1, max_tcu_num){
            //< if tcu active
                i[$] = i1[$] / 2; // parent of i1 and i2
                if ( (2*i[$]) == i1[$] ){ // came from ...
                    i2[$] = i1[$] + 1; // ...left
                }else{
                    i2[$] = i1[$] - 1; // ...right
                }

                count[i2[$]] = 0; // assume no active sibling(s)
            //> endif tcu active
            }

            /// <> fail spot 11

            spawn(1, max_tcu_num){
            //< if tcu active
                count[i1[$]] = subt[$]; // self is active
            //> endif tcu active
            }

            /// <> fail spot 12
```

```
            spawn(1, max_tcu_num){
            //< if tcu active
                subt[$] = count[i1[$]] + count[i2[$]]; // update total
                if ( i1[$] > i2[$] ){ // came from right
                    pnum[$] = pnum[$] + count[i2[$]];
                }

                i1[$] = i[$]; // advance to parent
            //> endif tcu active
            }

            /// <> fail spot 13

        }//end of "for" loop

        /// <> fail spot 14

        spawn(1, max_tcu_num){
        //< if tcu active
            count[1] = subt[$]; // record the total
        //> endif tcu active
        }
        //----- w 1   END ----------------------------------------------------
--

        /// <> fail spot 15


//#########################################################################
        //# W2:
        //# """
        //# private integers used:
        //#  >>> i, i1, i2 --parent and left/right child indices
        //#  >>> size --number of leaves at current node

//#########################################################################
        spawn(1, max_tcu_num){
        //< if tcu active
            i[$] = 1; // start at the root
            aux[1] = done[1];
        //> endif tcu active
        }
        size = n; // the whole tree is visible

        /// <> fail spot 16

        // traverse the tree from root to leaf
        while ( size != 1 ){

            /// <> fail spot 17

            spawn(1, max_tcu_num){
            //< if tcu active
                i1[$] = 2 * i[$]; // left
                i2[$] = i1[$] + 1; // right
                // compute accounted node values
                if ( (done[i1[$]] + done[i2[$]]) == 0 ){
                    aux[i1[$]] = 0;
                }else{
                    aux[i1[$]] = aux[i[$]] * done[i1[$]] / (done[i1[$]] +
done[i2[$]]);
                }
            //> endif tcu active
            }

            /// <> fail spot 18
```

```
            spawn(1, max_tcu_num){
            //< if tcu active
                aux[i2[$]] = aux[i[$]] - aux[i1[$]];
            //> endif tcu active
            }

            /// <> fail spot 19

            // processor allocation to left/right sub-trees
            spawn(1, max_tcu_num){
            //< if tcu active
                count[i1[$]] = count[i[$]] * ((size/2) - aux[i1[$]]) / (size -
aux[i[$]]);
            //> endif tcu active
            }

            /// <> fail spot 20

            spawn(1, max_tcu_num){
            //< if tcu active
                count[i2[$]] = count[i[$]] - count[i1[$]];
            //> endif tcu active
            }

            /// <> fail spot 21

            // go left/right based on processor number
            spawn(1, max_tcu_num){
            //< if tcu active
                if (pnum[$] <= count[i1[$]]){
                    i[$] = i1[$];
                }else{
                    i[$] = i2[$];
                    pnum[$] = pnum[$] - count[i1[$]];
                }
            //> endif tcu active
            }

            /// <> fail spot 22

            size = size / 2; // half of leaves
        }//end of "while" loop

        /// <> fail spot 23

        spawn(1, max_tcu_num){
        //< if tcu active
            k[$] = i[$] - (n - 1); // assign based on i
        //> endif tcu active
        }
        //----- w 2   END ----------------------------------------------------
--

        /// <> fail spot 24


//#############################################################################
        //# W3:

//#############################################################################
        spawn(1, max_tcu_num){
        //< if tcu active
            task[k[$]] = 1; // perform a task
        //> endif tcu active
        }
```

```
        //----- w 3   END ----------------------------------------------------
--

        /// <> fail spot 25


//#########################################################################
        //# W4:
        //# """
        //# private integers used:
        //#  >>> i, i1, i2 --parent and left/right child indices

//#########################################################################
        // mark the leaf done
        spawn(1, max_tcu_num){
        //< if tcu active
            i1[$] = k[$] + (n - 1); // leaf index
            done[i1[$]] = 1; // done!
        //> endif tcu active
        }

        /// <> fail spot 26

        // traverse the tree from leaf to root
        for (depth=1; depth <= log_n; depth++){

            /// <> fail spot 27

            spawn(1, max_tcu_num){
            //< if tcu active
                i[$] = i1[$] / 2;
                if ( (2*i[$]) == i1[$] ){ // came from ...
                    i2[$] = i1[$] + 1; // ...left
                }else{
                    i2[$] = i1[$] - 1; // ...right
                }

                // update progress tree and advance
                done[i[$]] = done[i1[$]] + done[i2[$]]; // update
                i1[$] = i[$]; // advance to parent
            //> endif tcu active
            }

            /// <> fail spot 28

        }
        //----- w 4   END ----------------------------------------------------
--

        /// <> fail spot 29

        #ifdef _cost
        sum=sum+(max_tcu_num - fail_index + 1);
        printf("%d alive=%d\tsum=%d\n", sum_c, max_tcu_num - fail_index + 1,
sum);
        sum_c++;
        #endif

    }// end of "while" loop - main
    //---- Main Loop   END ----------------------------------------------------
--

    // > > >   E N D   O F   M A I N   B O D Y   < < <

    #ifdef _cost
    printf("sum=%d\n",sum);
    #endif
```

```
    #ifdef _print_mem
    //appear on screen contents of shared arrays
    printf("\n task\n");
    for (depth=0; depth < n+1; depth++){
        printf(" %d", task[depth]);
    }printf("\n");

    printf("\n done\n");
    for (depth=0; depth < 2*n; depth++){
        printf(" %d", done[depth]);
    }printf("\n");

    printf("\n count\n");
    for (depth=0; depth < 2*n; depth++){
        printf(" %d", count[depth]);
    }printf("\n");

    printf("\n aux\n");
    for (depth=0; depth < 2*n; depth++){
        printf(" %d", aux[depth]);
    }printf("\n");

    printf("\n k\n");
    for (depth=0; depth < n+1; depth++){
        printf(" %d", k[depth]);
    }printf("\n");

    printf("\n i\n");
    for (depth=0; depth < n+1; depth++){
        printf(" %d", i[depth]);
    }printf("\n");

    printf("\n i1\n");
    for (depth=0; depth < n+1; depth++){
        printf(" %d", i1[depth]);
    }printf("\n");

    printf("\n i2\n");
    for (depth=0; depth < n+1; depth++){
        printf(" %d", i2[depth]);
    }printf("\n");

    printf("\n pnum\n");
    for (depth=0; depth < n+1; depth++){
        printf(" %d", pnum[depth]);
    }printf("\n");

    printf("\n subt\n");
    for (depth=0; depth < n+1; depth++){
        printf(" %d", subt[depth]);
    }printf("\n");

    printf("\n size\n %d\n", size);

    printf("\n tcu_is_active\n");
    for (depth=0; depth < n+1; depth++){
        printf(" %d", tcu_is_active[depth]);
    }printf("\n");

    printf("\n fps_mode\n %d\n", fps_mode);
    printf("\n fail_index\n %d\n", fail_index);

    printf("\n");
    #endif
}
```

B.8 Αναλυτικές Μετρήσεις Προσομοιώσεων Υλοποίησης Αλγόριθμου W

n	sim #1	sim #2	sim #3	sim #4	sim #5		min	max	sum	(sum –min -max) / 3	round
2	6.762	6.759	6.759	6.762	6.759		6.759	6.762	33.801	6.760,00	6.760
4	17.720	17.714	17.600	17.752	17.621		17.600	17.752	88.407	17.685,00	17.685
8	34.533	34.380	34.400	34.524	34.406		34.380	34.533	172.243	34.443,33	34.443
16	58.405	58.105	58.218	58.609	58.143		58.105	58.609	291.480	58.255,33	58.255
32	88.035	87.987	89.530	89.737	89.337		87.987	89.737	444.626	88.967,33	88.967
64	127.417	127.722	127.872	127.497	128.386		127.417	128.386	638.894	127.697,00	127.697
128	178.311	177.410	177.765	178.100	178.479		177.410	178.479	890.065	178.058,67	178.059
256	247.155	247.507	247.813	245.493	245.522		245.493	247.813	1.233.490	246.728,00	246.728
512	348.146	347.773	348.187	348.124	348.068		347.773	348.187	1.740.298	348.112,67	348.113
1024	510.443	510.146	510.586	509.430	509.598		509.430	510.586	2.550.203	510.062,33	510.062

Πίνακας B.1 – Μετρήσεις clock cycles Σεναρίου Υποεκτίμησης $f = {}^{n}/_{2}$ (fail spots 3 and 25 are active)

n	sim #1	sim #2	sim #3	sim #4	sim #5		min	max	sum	(sum –min -max) / 3	round
2	6.762	6.759	6.759	6.762	6.762		6.759	6.762	33.804	6.761,00	6.761
4	10.481	10.477	10.460	10.475	10.464		10.460	10.481	52.357	10.472,00	10.472
8	13.983	13.964	13.983	13.961	13.882		13.882	13.983	69.773	13.969,33	13.969
16	18.154	18.276	18.300	18.194	18.234		18.154	18.300	91.158	18.234,67	18.235
32	23.451	23.539	23.490	23.555	23.383		23.383	23.555	117.418	23.493,33	23.493
64	30.676	30.547	30.605	53.381	30.674		30.547	53.381	175.883	30.651,67	30.652
128	72.596	72.889	72.683	73.053	72.722		72.596	73.053	363.943	72.764,67	72.765
256	104.268	106.907	106.548	106.577	59.537		59.537	106.907	483.837	105.797,67	105.798
512	172.996	166.029	171.913	161.645	160.118		160.118	172.996	832.701	166.529,00	166.529
1024	284.789	279.681	270.328	273.640	277.170		270.328	284.789	1.385.608	276.830,33	276.830

Πίνακας B.2 - Μετρήσεις clock cycles Σεναρίου Υποεκτίμησης $f = {}^{n}/_{4}$ (fail spots 3 and 25 are active)

(συνέχεια στην επόμενη σελίδα)

n	sim #1	sim #2	sim #3	sim #4	sim #5		min	max	sum	(sum –min -max) / 3	round
2	6.743	6.743	6.746	6.743	6.743		6.743	6.746	33.718	6.743,00	6.743
4	17.641	17.664	17.646	17.795	17.664		17.641	17.795	88.410	17.658,00	17.658
8	34.314	34.341	34.454	34.327	34.163		34.163	34.454	171.599	34.327,33	34.327
16	58.077	57.805	57.436	57.291	57.817		57.291	58.077	288.426	57.686,00	57.686
32	87.412	88.743	87.527	87.172	87.942		87.172	88.743	438.796	87.627,00	87.627
64	125.504	125.331	126.679	127.779	125.998		125.331	127.779	631.291	126.060,33	126.060
128	173.682	173.698	174.031	173.661	174.078		173.661	174.078	869.150	173.803,67	173.804
256	240.509	239.813	240.391	238.906	241.065		238.906	241.065	1.200.684	240.237,67	240.238
512	332.600	333.112	332.099	331.396	331.503		331.396	333.112	1.660.710	332.067,33	332.067
1024	477.827	477.319	478.138	477.931	477.133		477.133	478.138	2.388.348	477.692,33	477.692

Πίνακας B.3 - Μετρήσεις clock cycles Σεναρίου Υπερεκτίμησης f $= {}^{n}/_{2}$ (fail spots 1 and 15 are active)

n	sim #1	sim #2	sim #3	sim #4	sim #5		min	max	sum	(sum –min -max) / 3	round
2	6.747	6.747	6.750	6.747	6.747		6.747	6.750	33.738	6.747,00	6.747
4	10.443	10.529	10.453	10.515	10.451		10.443	10.529	52.391	10.473,00	10.473
8	13.955	13.925	13.795	13.951	13.870		13.795	13.955	69.496	13.915,33	13.915
16	18.032	17.900	17.902	17.893	17.990		17.893	18.032	89.717	17.930,67	17.931
32	40.459	23.217	22.854	40.399	23.226		22.854	40.459	150.155	28.947,33	28.947
64	29.675	29.756	29.771	29.948	29.710		29.675	29.948	148.860	29.745,67	29.746
128	39.856	69.232	69.589	69.685	39.780		39.780	69.685	288.142	59.559,00	59.559
256	96.726	99.591	56.632	99.855	56.533		56.533	99.855	409.337	84.316,33	84.316
512	87.997	145.505	148.779	147.440	157.824		87.997	157.824	687.545	147.241,33	147.241
1024	248.153	246.985	244.766	247.406	252.493		244.766	252.493	1.239.803	247.514,67	247.515

Πίνακας B.4 - Μετρήσεις clock cycles Σεναρίου Υπερεκτίμησης $f = {}^{n}/_{4}$ (fail spots 1 and 15 are active)

(συνέχεια στην επόμενη σελίδα)

n	sim #1		sim #2		sim #3		sim #4		sim #5	
	άθροισμα	βήματα	άθροισμα	βήματα	άθροισμα	βήματα	άθροισμα	βήματα	άθροισμα	βήματα
4	4	3	4	3	4	3	4	3	4	3
8	8	4	8	4	8	4	8	4	8	4
16	16	5	16	5	16	5	16	5	16	5
32	32	6	32	6	32	6	32	6	32	6
64	64	7	64	7	64	7	64	7	64	7
128	128	8	128	8	128	8	128	8	128	8
256	256	9	256	9	256	9	256	9	256	9
512	512	10	512	10	512	10	512	10	512	10
1024	1024	11	1024	11	1024	11	1024	11	1024	11

Πίνακας B.5 - Μετρήσεις αθροίσματος και βημάτων Σεναρίου Υποεκτίμησης f = $^{n}/_{2}$ (fail spots 3 and 25)

n	sim #1		sim #2		sim #3		sim #4		sim #5	
	άθροισμα	βήματα	άθροισμα	βήματα	άθροισμα	βήματα	άθροισμα	βήματα	άθροισμα	βήματα
4	5	2	5	2	5	2	5	2	5	2
8	11	2	11	2	11	2	11	2	11	2
16	21	2	21	2	21	2	21	2	21	2
32	46	2	42	2	56	3	42	2	42	2
64	84	2	84	2	84	2	84	2	84	2
128	168	2	222	3	168	2	168	2	222	3
256	444	3	444	3	444	3	444	3	336	2
512	888	3	888	3	888	3	888	3	888	3
1024	1776	3	1776	3	1776	3	1776	3	1776	3

Πίνακας B.6 - Μετρήσεις αθροίσματος και βημάτων Σεναρίου Υποεκτίμησης f = $^{n}/_{4}$ (fail spots 3 and 25)

(συνέχεια στην επόμενη σελίδα)

n	sim #1		sim #2		sim #3		sim #4		sim #5	
	άθροισμα	βήματα	άθροισμα	βήματα	άθροισμα	βήματα	άθροισμα	βήματα	άθροισμα	βήματα
4	4	3	4	3	4	3	4	3	4	3
8	8	4	8	4	8	4	8	4	8	4
16	16	5	16	5	16	5	16	5	16	5
32	32	6	32	6	32	6	32	6	32	6
64	64	7	64	7	64	7	64	7	64	7
128	128	8	128	8	128	8	128	8	128	8
256	256	9	256	9	256	9	256	9	256	9
512	512	10	512	10	512	10	512	10	512	10
1024	1024	11	1024	11	1024	11	1024	11	1024	11

Πίνακας B.7 - Μετρήσεις αθροίσματος και βημάτων Σεναρίου Υπερεκτίμησης $f = n/2$ (fail spots 1 and 15)

n	sim #1		sim #2		sim #3		sim #4		sim #5	
	άθροισμα	βήματα	άθροισμα	βήματα	άθροισμα	βήματα	άθροισμα	βήματα	άθροισμα	βήματα
4	5	2	5	2	5	2	5	2	5	2
8	11	2	11	2	11	2	11	2	11	2
16	21	2	21	2	21	2	21	2	21	2
32	42	2	42	2	42	2	42	2	42	2
64	111	3	84	2	84	2	84	2	84	2
128	222	3	222	3	222	3	222	3	168	2
256	444	3	336	2	444	3	444	3	444	3
512	888	3	888	3	888	3	888	3	888	3
1024	1776	3	1776	3	1776	3	1776	3	1776	3

Πίνακας B.8 - Μετρήσεις αθροίσματος και βημάτων Σεναρίου Υπερεκτίμησης $f = n/4$ (fail spots 1 and 15)

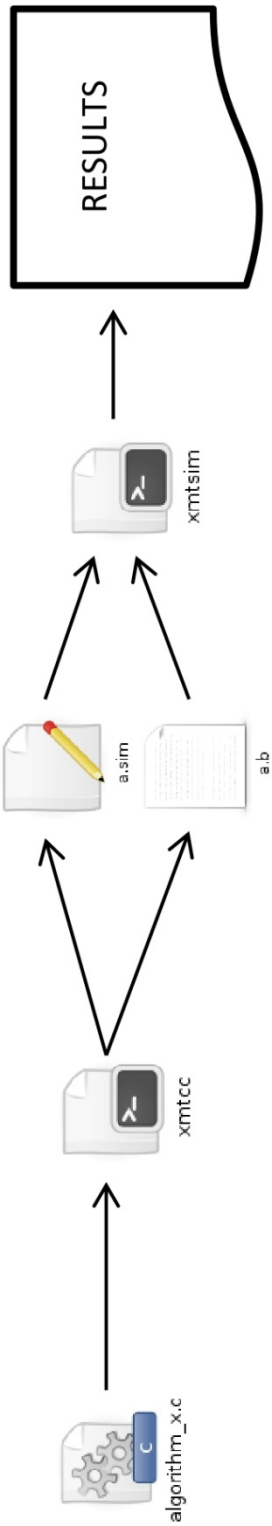
Παράρτημα Γ

Γ.1 Οδηγός Εκτέλεσης Υλοποίησης Αλγορίθμου X

Προκειμένου να εκτελέσετε την υλοποίηση του αλγορίθμου X θα πρέπει να εκτελέσετε την ακόλουθη εντολή:

```
Terminal$ ./xmt_program.sh algorithm_x
```

Γ.2 Ροή Δεδομένων για “xmt_program.sh”



Γ.3 Bash Script “xmt_program.sh”

```
#!/bin/bash

#    _____________________________________________________
#   {_____   |##########################################|   _____}
#         \  |###                                    ###|  /
#       __/  |### Name : Efstathios Ioakim           ###|  \__
#      {__   |### Year : 2012                        ###|   __}
#         \  |###                                    ###|  /
#    _____/  |###                                    ###|  \_____
#   {________|##########################################|________}
#            `````````````````````````````````````````

#
# PROGRAMMER COMMENTS:
#
# Script Main Task:
# """""""""""""""""""
# This BASH(.sh) script is used for compiling and simulating xmtc code
#

################################################################################
##
### SCRIPT BEGINS
#####################################################################
################################################################################
##
#clear contents onscreen (terminal)
clear

#set params flag to zero
check_flag=0

#check if parameter was given by user
if [ $1 = "algorithm_x" ]; then
    #set program name equal to parameter
    program_name="algorithm_x.c"
    #set parameter detect flag to TRUE
    check_flag=1
fi

#check if given parameter is what expected (check via flag)
# => if TRUE do compiling and simulation
# => if FALSE appear error message
if [ $check_flag = 1 ]; then
    #compile the xmtc code
    xmtcc $program_name
    #begin simulation of program
    xmtsim -cycle -binload a.b a.sim
else
    #appear bad parameter error message on screen
    echo "Error : Bad parameter!"
fi
################################################################################
##
### SCRIPT ENDS
#####################################################################
################################################################################
##
```

Γ.4 Κώδικας XMTC Αλγορίθμου X (algorithm_x.c)

```
#include <xmtc.h>

//    _____________________________________________________________
//   {_____   |#########################################|   _____}
//         \  |###                                   ###|  /
//       __/  |### Name : Efstathios Ioakim           ###|  \__
//      {__   |### Year : 2012                        ###|   __}
//         \  |###                                   ###|  /
//    _____/  |###                                   ###|  \_____
//   {________|#########################################|________}
//            `````````````````````````````````````````
//

//#############################################################################
##
//# G L O B A L    V A R I A B L E S
##########################################
//#############################################################################
##
#define _n_512 0
//#define _print_mem 0

#if defined _n_4
    #define n 4
    #define log_n 2

#elif defined _n_8
    #define n 8
    #define log_n 3

#elif defined _n_16
    #define n 16
    #define log_n 4

#elif defined _n_32
    #define n 32
    #define log_n 5

#elif defined _n_64
    #define n 64
    #define log_n 6

#elif defined _n_128
    #define n 128
    #define log_n 7

#elif defined _n_256
    #define n 256
    #define log_n 8

#elif defined _n_512
    #define n 512
    #define log_n 9

#elif defined _n_1024
    #define n 1024
    #define log_n 10

#elif defined _n_2048
    #define n 2048
    #define log_n 11

#elif defined _n_4096
```

```
    #define n 4096
    #define log_n 12

#elif defined _n_8192
    #define n 8192
    #define log_n 13

#elif defined _n_16384
    #define n 16384
    #define log_n 14

#elif defined _n_32768
    #define n 32768
    #define log_n 15

#elif defined _n_65536
    #define n 65536
    #define log_n 16

#elif defined _n_131072
    #define n 131072
    #define log_n 17

#endif

int main(){

//#########################################################################
    //# shared integer arrays

//#########################################################################
    int task[n+1];  //task[1..n]
    int done[2*n];  //done[1..2n-1]
    //-------------------------------------------------------------------------
--


//#########################################################################
    //# miscellaneous variables

//#########################################################################
    int max_tcu_num; // int usded for number of available processors (TCUs)
    int pr_var; // int used for printing messages
    int log2[n+1]; // log2(PID) for each tcu
    //-------------------------------------------------------------------------
--


//#########################################################################
    //# shared integer array initialization

//#########################################################################
    spawn(0, n){
        task[$] = 0;    //task[n+1]
    }
    spawn(0, 2*n-1){
        done[$] = 0;    //done[2n]
    }
    //-------------------------------------------------------------------------
--


//#########################################################################
    //# miscellaneous variables initialization

//#########################################################################
    max_tcu_num = n; // initial number of available processors (TCUs)
```

```
    spawn(0, n){ // calculate log2(PID)
        int tmp_num;
        log2[$] = 0;
        tmp_num = $;
        while( tmp_num >>= 1 ){
            log2[$]+=1;
        }
    }
    //-------------------------------------------------------------------------
--


    // > > >   B E G I N   O F   M A I N   B O D Y   < < <


//#############################################################################
    //# algorithm x main body

//#############################################################################
    spawn(1, max_tcu_num){
        int where, done_int;    // current node index and its "done" value
        int left, right;    // left/right child values in progress tree
        int tmp_num;
        int log_where;
        int PID_bit;

        where = n + $ - 1; // initial assignment at the leafs

        while ( done[1] != 1 ){ // as long as the root is not marked as done

            done_int = done[where]; //doneness of this subtree

            if ( done_int ){ // begin of "if else if..." level 1

                where = where / 2; // move up one level

            }else if ( !(done_int) && (where > (n-1)) ){ // at a leaf

                if ( task[where-n+1] == 0 ){
                    task[where-n+1] = 1; // perform Do-All task
                }else if ( task[where-n+1] == 1 ){
                    done[where] = 1; // indicate "done"
                }

            }else if ( !(done_int) && (where <= (n-1)) ){ // interior tree
node

                left = done[2*where]; // left child value
                right = done[2*where+1]; // right child value

                if ( left && right ){
                    done[where] = 1; // both children done
                }else if ( !(left) && right ){
                    where = 2*where; // move left
                }else if ( left && !(right) ){
                    where = 2*where+1; // move right
                }else if ( !(left) && !(right) ){
                    // calculate log_where
                    log_where = 0;
                    tmp_num = where;
                    while( tmp_num >>= 1 ){ // log(where) = msb position
                        log_where++;
                    }
                    // calculate the value of PID bit
                    PID_bit = ($) & (1<<(log2[$]-log_where));
```

```
                    // move down according to the PID bit
                    if ( !(PID_bit) ){
                        where = 2*where; // move left
                    }else if ( PID_bit ){
                        where = 2*where+1; // move right
                    }
                }

            }// end of "if else if..." level 1

        }//end of "while" loop

    }// end of spawn()
    //---------------------------------------------------------------------------
--

    // > > >   E N D   O F   M A I N   B O D Y   < < <

    #ifdef _print_mem
    //appear on screen contents of shared arrays
    printf("\n task\n");
    for (pr_var=0; pr_var < n+1; pr_var++){
        printf(" %d", task[pr_var]);
    }printf("\n");

    printf("\n done\n");
    for (pr_var=0; pr_var < 2*n; pr_var++){
        printf(" %d", done[pr_var]);
    }printf("\n");

    #endif
}
```