ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ
ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Μελέτη και Αξιολόγηση Transactional Memory σε αλγορίθμους άμορφου παραλληλισμού

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Κωνσταντίνος Β. Καζατζής

Επιβλέπων: Ν. Κοζύρης
Καθηγητής Ε.Μ.Π

Αθήνα, Ιούλιος 2017

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Μελέτη και Αξιολόγηση Transactional Memory σε αλγορίθμους άμορφου παραλληλισμού

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Κωνσταντίνος Β. Καζατζής

Επιβλέπων: Ν. Κοζύρης
Καθηγητής Ε.Μ.Π

Εγκρίθηκε από την τριμελή επιτροπή την 12η Ιουλίου 2017

…….	…….	…….
Γ. Γκούμας	Ν. Κοζύρης	Δ.Τσουμάκος
Επίκουρος Καθηγητής Ε.Μ.Π	Καθηγητής Ε.Μ.Π	Αναπληρωτής Καθηγητής Ε.Μ.Π

Αθήνα, Ιούλιος 2017

……
Κωνσταντίνος Β. Καζατζής
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Οι πολυπύρηνες αρχιτεκτονικές είναι ο κυρίαρχος τρόπος σχεδιασμού επεξεργαστών σήμερα. Οι νέοι επεξεργαστές κατασκευάζονται με όλο και αυξανόμενο αριθμό πυρήνων καθιστώντας τα υπολογιστικά συστήματα ικανά για επεξεργασία δεκάδων νημάτων. Ο προγραμματισμός ενός πολυπύρηνου συστήματος δεν είναι ίδιος σε σχέση με έναν μονοπύρηνο. Για να είναι δυνατή η πλήρης αξιοποίηση του διαθέσιμου υλικού, ο αλγόριθμος πρέπει να κάνει σαφείς διακρίσεις σε ανεξάρτητες εργασίες προς εκτέλεση. Διαφορετικοί αλγόριθμοι έχουν και διαφορετική παράλληλη συμπεριφορά, με ένα κύριο χαρακτηριστικό να είναι πόσο ευκολά υλοποιείται ο συγχρονισμός στο εκτελούμενο πρόγραμμα.
Στην εργασία αυτή θα μελετηθούν αλγόριθμοι γράφων που παρουσιάζουν άμορφη πρόσβαση στα κοινά δεδομένα, υλοποιούμενοι σε worklist λογική για την αξιοποίηση του παραλληλισμού τους. Ο συγχρονισμός θα υλοποιηθεί μέσω Transactional Memory, και θα συγκριθεί με μια software lock-free τεχνική που υλοποιείται στο σύστημα Galois, πάνω στο οποίο θα γίνει και η εκτέλεση τους. Τέλος, θα αναλυθεί και θα σχολιαστεί η απόδοση τους σε πολυπύρηνα συστήματα, σε σχέση με την τεχνική του Galois.

Λέξεις Κλειδιά:
Παραλληλισμός στα δεδομένα, λίστα εργασιών, Transactional Memory, Galois, semantic commutativity, πολυπύρηνες αρχιτεκτονικές, επιτάχυνση εκτέλεσης

Abstract

Multithreaded architectures are the most dominant way of processor designing nowadays. The number of cores in computers processors keeps increasing which creates systems capable of running tenths of threads at the same time. Programming such a system is much different from a single core computer. In order to be able to exploit the capabilities of the hardware to the fullest, the program has to make clear distinctions between different tasks. Different algorithms have different parallel characteristics, with one of them being how easy it is to implement synchronization.

In this paper, we will discuss and analyze algorithms upon graphs, where the shared data access is performed in an amorphous way, when implemented in a worklist logic to exploit data parallelism. The synchronization will be implemented with Transactional Memory, and it will be compared with the software lock-free technique used on the Galois system that will be the runtime system. Finally, we will discuss their speedup performance in comparison with Galois, when run on multicore systems.

Key words

Data parallelism, worklist, Transactional Memory, Galois, semantic commutativity, multithreads architectures, execution speedup

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τους καθηγητές κ. Γεώργιο Γκούμα και κ. Νεκτάριο Κοζύρη για τις πολύτιμές γνώσεις και τα εναύσματα αναζήτησης που μου πρόσφεραν στις διαλέξεις τους.

Ευχαριστώ ιδιαίτερα τον Υποψήφιο Διδάκτωρ Δημήτριο Σιακαβάρα για την αμέριστη βοήθεια του στην συγγραφή της διπλωματικής εργασίας,

Τέλος, ευχαριστώ την οικογένεια μου για την στήριξη και συμπαράσταση σε όλη την διάρκεια των σπουδών μου.

Περιεχόμενα

Λίστα εικόνων

1 Εισαγωγή

Το κύριο χαρακτηριστικό σύγκρισης της επίδοσης δύο επεξεργαστών είναι η ταχύτητα με την οποία εκτελούν εντολές στην μονάδα του χρόνου ή καλύτερα σε έναν κύκλο του ρολογιού τους. Η απόδοση αυτή είναι συνάρτηση πολλών χαρακτηριστικών της αρχιτεκτονικής του επεξεργαστή και μέχρι και το 2001 περιοριζόταν σε ένα πυρήνα ικανό να εκτελέσει υπολογισμούς. Ο πρώτος διπύρηνος επεξεργαστής έκανε την εμφάνιση του το 2001 και ήταν ο POWER4 από την IBM. Ωστόσο ήταν το 2005 που η AMD, μια εταιρεία που φτιάχνει επεξεργαστές για το ευρύ κοινό, εμφάνισε τον Athlon 64 X2 που ήτανε ο πρώτος διπύρηνος επεξεργαστής διαθέσιμος στον μέσο καταναλωτή. Από τότε οι πολυπύρηνοι επεξεργαστές έγιναν ο κύριος στόχος της αγοράς, με τις εταιρείες κατασκευής να βρίσκουν σε αυτούς έναν τρόπο να συνεχίσουν την κυριαρχία του νόμου του Moore, καθώς ο αριθμός των τρανζίστορ σε ένα κύκλωμα συνέχιζε να αυξάνεται.

Η αύξηση της απόδοσης ενός επεξεργαστή έχει συνήθως άμεσο αντίκτυπο στον χρόνο εκτέλεσης των προγραμμάτων. Ο διπλασιασμός του IPC (εντολές ανά κύκλο επεξεργαστή) μπορεί να κάνει ένα πρόγραμμα να τρέξει σχεδόν στον μισό χρόνο, σε σχέση με ένα άλλο σύστημα, χωρίς να χρειάζεται καμία παρέμβαση από τον προγραμματιστή. Η αύξηση του IPC όμως σχετίζεται με πολλούς παράγοντες και ο πιο προφανής από αυτούς είναι η συχνότητα λειτουργίας του επεξεργαστή. Και ενώ η μικροηλεκτρονική έδωσε την δυνατότητα για καλύτερα κυκλώματα, από άποψη υλικών, σχεδιασμού και ποσότητας τρανζίστορ, η αύξηση της συχνότητας λειτουργίας έφτασε ένα πάνω όριο. Η κατανάλωση ενέργειας ενός επεξεργαστή είναι άμεση συνάρτηση της συχνότητας λειτουργίας του και αυτό απαιτεί και αυξημένες ανάγκες στην τροφοδοσία του συστήματος αλλά και έξυπνους τρόπους ψύξης του. Ο ‘’τοίχος΄΄ αυτός στην απόδοση του ενός πυρήνα οδήγησε στον σχεδιασμό των πολυπύρηνων αρχιτεκτονικών. Δεδομένου λοιπόν ότι ο αριθμός των πυρήνων σε έναν επεξεργαστή αναμένεται ότι θα συνεχίσει να αυξάνεται, για να καταστεί δυνατή η εκμετάλλευση των τρανζίστορ, υπάρχει μια μετάβαση στο πρόβλημα της γρηγορότερης εκτέλεσης ενός προγράμματος από το υλικό στον προγραμματιστή.

Σε αντίθεση με έναν μονοπύρηνο επεξεργαστή, που στην αριθμητική λογική μονάδα του έχει πολλά διαφορετικά αδιαφανή μονοπάτια για εκτέλεση εντολών εκτός σειράς, ο προγραμματισμός ενός πολυπύρηνου συστήματος απαιτεί προγραμματιστικό κόπο. Όταν ο προγραμματιστής γράφει σειριακό κώδικά, η εκτέλεση γίνεται σε έναν πυρήνα υπολογισμού, ο οποίος όμως είναι σχεδιασμένος να βρίσκει δυνατότητες παραλληλισμού στην εκτέλεση των εντολών μέσω διαφόρων τεχνικών. Αυτές οι τεχνικές όμως δεν είναι ορατές στον προγραμματιστή, σε αντίθεση με τα πολυπύρηνα συστήματα όπου ο κώδικας πρέπει να κάνει σαφείς διακρίσεις σε παράλληλες εργασίες για να μπορέσει να εκμεταλλευτεί το διαθέσιμο υλικό.

Ένα ακόμη πρόβλημα που μπαίνει σε προγράμματα με παράλληλα κομμάτια κώδικά είναι ο συγχρονισμός. Ανάλογα με την εφαρμογή, ο προγραμματισμός ενός παράλληλου αλγορίθμου πρέπει να έχει κάποιες ή και όλες από τις παρακάτω ιδιότητες: Ατομικότητα (Είτε μια εργασία θα εκτελεστεί επιτυχώς είτε καθόλου), Συνέπεια (Ότι το κοινό μέσο αποθήκευσης είτε είναι βάση δεδομένων είτε είναι δομή δεδομένων στην μνήμη δεν θα βρεθεί ποτέ σε ασυνεπή κατάσταση), Απομόνωση (Μια εργασία πρέπει να τρέχει σε ένα σύστημα όπου θα φέρει το ίδιο αποτέλεσμα σε σχέση με ένα σύστημα στο οποίο θα έτρεχε μόνη της) και Αντοχή (Μια εργασία θα τελειώσει την εκτέλεση της και τα αποτελέσματα αυτής θα παραμείνουν διαθέσιμα).

Στην διπλωματική αυτή εργασία, θα μελετηθεί ένα τρόπος παραλληλισμού αλγορίθμων ‘’ακανόνιστων’’ προγραμμάτων (irregular programs) σε αντίθεση με τα ‘’κανονικά΄΄ προγράμματα (regular programs), το Galois ως σύστημα εκτέλεσης τέτοιων αλγορίθμων για εκμετάλλευση της ακανόνιστης πρόσβασης τους στα κοινά δεδομένα (amorphous data-parallelism). Τέλος θα μελετηθεί η απόδοση διαφόρων irregular

αλγορίθμων σε πολυπύρηνες αρχιτεκτονικές κοινής μνήμης πάνω στο σύστημα Galois σε σχέση με την υλοποίηση τους χρησιμοποιώντας Transactional Memory για συγχρονισμό.

2 Amorphous data-parallelism

Για την παραλληλοποίηση ενός αλγορίθμου, ο προγραμματιστής σχεδόν πάντα πρέπει να βρει εργασίες που μπορούν να εκτελεστούν παράλληλα, πράγμα το οποίο σημαίνει ότι η πρόσβαση των εργασιών αυτών στα κοινά δεδομένα δεν γίνεται ταυτόχρονα [1]. Η αξιοποίηση του παραλληλισμού ενός αλγορίθμου σύμφωνα με αυτήν την προσέγγιση μπορεί να αναπαρασταθεί ως ένας αριθμός επαναλήψεων, όπου η κάθε μια ασκεί την ίδια ακριβώς δουλειά σε ένα διαφορετικό κομμάτι των κοινών δεδομένων. Η αναπαράσταση αυτή ονομάζεται Single Instruction Multiple Data (SIMD). Σε αυτήν την αναπαράσταση τα κομμάτια των δεδομένων πάνω στα οποία εκτελούνται οι εργασίες πρέπει να είναι εντελώς ανεξάρτητα μεταξύ τους, αλλιώς υπάρχουν καταστάσεις συναγωνισμού (race conditions) τις οποίες καλείται να λύσει η μέθοδος συγχρονισμού. Όταν λοιπόν θέλουμε να παραλληλοποίησουμε ένα αλγόριθμο πρέπει i) να βρούμε εργασίες που μπορούν να εκτελεστούν ταυτόχρονα και στην συνέχεια ii) να εξετάσουμε αν υπάρχουν καταστάσεις συναγωνισμού.

2.1 Παραλληλισμός δεδομένων σε regular προγράμματα

Με τον όρο ‘’κανονικά’’ προγράμματα, που από εδώ και πέρα θα αναφέρονται ως regular programs, εννοούνται προγράμματα που στον πυρήνα τους έχουν την επεξεργασία δομών δεδομένων όπως οι πίνακες. Αλγόριθμοι που εφαρμόζονται πάνω σε πίνακες, όπως οι πράξεις πολλαπλασιασμού, πρόσθεσης πινάκων στην γραμμική άλγεβρα, είναι εξ-ορισμού παράλληλοι στα δεδομένα. Η κύρια μέθοδος παραλληλισμού είναι η εύρεση των επαναλήψεων που μπορούν να εκτελεστούν ταυτόχρονα και η εφαρμογή δομών όπως η *parallel-for*. Τέτοιες δομές υπάρχουν σε πάρα πολλές γλώσσες και frameworks από την Fortran [2], OpenMP [3] μέχρι και την Golang της Google [4].

Για να μετατραπεί ο σειριακός αλγόριθμος σε παράλληλο, το μόνο που έχει να κάνει ο προγραμματιστής είναι να βρει την *for-loop* επανάληψη που είναι παραλληλοποιήσιμη και να την μετατρέψει σε *parallel-for*, μια διαδικασία που διευκολύνεται με εργαλεία ανάλυσης κώδικα [5] για την εύρεση των εν λόγω επαναλήψεων.

2.2 Παραλληλισμός δεδομένων σε irregular προγράμματα

Σε αντίθεση με τους regular αλγορίθμους, τα ‘’ακανόνιστα’’ προγράμματα, που από εδώ και πέρα θα αναφέρονται ως irregular προγράμματα, δεν έχουν ως κύριο χαρακτηριστικό την επεξεργασία δομών όπως οι πίνακες. Οι δομές δεδομένων των irregular αλγορίθμων είναι κυρίως pointer-based με λίστες, γράφους και δέντρα. Η παραλληλοποίηση τέτοιων αλγορίθμων δεν μπορεί να βασίζεται στις προηγούμενες μεθόδους καθώς η πρόσβαση στα κοινά δεδομένα δεν γίνεται με δομημένο τρόπο. Οι παράλληλες εργασίες που υπάρχουν, έχουν πολλές φορές race conditions, το οποίο απαιτεί συγχρονισμό. Ωστόσο υπάρχει αρκετός χώρος παραλληλισμού ο οποίος όμως είναι αρκετά δυσκολότερο να βρεθεί και να αξιοποιηθεί. Ένας από τους τρόπους που έχει προταθεί και χρησιμοποιείται σε αυτήν την εργασία εκτενώς είναι λίστες εργασιών (worklists)[1] [6] [7].

Ας πάρουμε για παράδειγμα έναν αλγόριθμο που εφαρμόζεται σε έναν τριγωνοποιημένο γράφο. Ο αλγόριθμος Delaunay Mesh Refinement έχει ως είσοδο ένα γράφο όπου οι κόμβοι έχουν συνδέσεις του σχηματίζουν τρίγωνα. Στον αλγόριθμό αυτό ορίζουμε μια ιδιότητα του τριγώνου και λέμε ότι όποια τρίγωνα έχουν αυτήν την ιδιότητα είναι ανεπιθύμητα ή ‘’κακά’’.

[1] Οι worklists θα συζητηθούν εκτενώς σε επόμενο κεφάλαιο

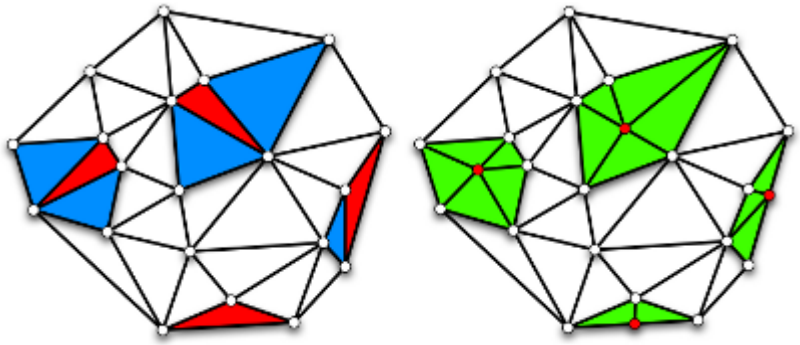

Figure 1: Αριστερά γράφος με ΄΄κακά΄΄ τρίγωνα, δεξιά γράφος refined

Για παράδειγμα ας ορίσουμε ως κακά τρίγωνα αυτά που έχουν γωνίες κάτω των 30 μοιρών. Αυτά είναι τα κόκκινα τρίγωνα όπως φαίνονται στην Figure 1. Ο αλγόριθμος Delaunay refinement χρησιμοποιείται για να σβήσει τα τρίγωνα αυτά από τον γράφο, και να δημιουργήσει άλλα χωρίς την ιδιότητα του ΄΄κακού΄΄. Ο τρόπος με τον οποίο γίνεται αυτό είναι να εισάγονται τα ανεπιθύμητα τρίγωνα σε μια worklist, να ανατίθεται κάθε στοιχείο της worklist σε έναν εργάτη, και κάθε εργάτης να εφαρμόζει έναν αλγόριθμο πάνω στο τρίγωνο. Αν κάποιο από τα νέα δημιουργημένα τρίγωνα είναι ΄΄κακό΄΄ τότε θα μπει εκ νέου στην worklist. Είναι προφανές, ότι η διαδικασία της τριγωνοποίησης που εφαρμόζεται σε κάθε τρίγωνο διαβάζει και γράφει στα κοινά δεδομένα, τα οποία είναι οι κόμβοι και οι συνδέσεις του γράφου. Είναι επίσης προφανές ότι σίγουρα θα υπάρχουν τρίγωνα όπου δεν θα υπάρχει πρόβλημα εξαρτήσεων, όπως φαίνεται και από την εικόνα, καθώς μπορεί να μην γειτονεύουν. Φαίνεται λοιπόν ότι: i) υπάρχει αρκετός χώρος για παραλληλισμό του αλγορίθμου αλλά και ότι ii) οι προσβάσεις στην μνήμη δεν μπορούν να πέσουν κάτω από μια δομημένη μορφή όπως στους πίνακες. Αυτά τα δύο στοιχεία είναι τα κύρια χαρακτηριστικά των irregular αλγορίθμων.

Αξίζει να σημειωθεί εδώ ότι ο αλγόριθμος που χρησιμοποιείται για να εξομαλύνει ένα ΄΄κακό΄΄ τρίγωνο μοιάζει αρκετά με ένα Transaction πάνω στα κοινά δεδομένα που εδώ δεν είναι μια βάση δεδομένων αλλά μια δομή στην μνήμη. Σε παρακάτω ενότητα θα συζητηθεί εκτενώς ο αλγόριθμος Delaunay mesh refinement, καθώς και οι υπόλοιποι αλγόριθμοι που θα χρησιμοποιηθούν.

2.2.1 Τρόποι παραλληλισμού irregular αλγορίθμων

Στατικές Μέθοδοι: Η πιο εύκολη προσέγγιση είναι να χρησιμοποιηθεί κάποιο εργαλείο ανάλυσης του κώδικα για *points-to analysis* και *shape analysis* [8] [9]. Με αυτό τον τρόπο μπορούν να βρεθούν κομμάτια του κώδικα που είναι ανεξάρτητα μεταξύ τους. Αυτή η μέθοδος όμως είναι compiler-based. Αυτό σημαίνει ότι το τελικό πρόγραμμα δεν έχει κάποιο τρόπο αλλαγής της συμπεριφοράς του ανάλογα με την είσοδο. Δηλαδή, ο παραλληλισμός ο οποίος προκύπτει από την πρόσβαση στα κοινά δεδομένα, ο οποίος είναι και ο μόνος στους irregular αλγορίθμους δεν μπορεί να αξιοποιηθεί στο έπακρο.

Ημι-στατικές Μέθοδοι: Η κύρια προσέγγιση εδώ είναι η *inspector-executor* [10]. Σε αυτήν την μέθοδο ο υπολογισμός χωρίζεται σε δυο φάσεις. Στην πρώτη φάση γίνεται αναγνώριση των εξαρτήσεων μεταξύ των εργασιών που πρέπει να εκτελεστούν (inspector), και στην δεύτερη φάση εκτελούνται οι δουλειές παράλληλα (executor). Για να μπορέσει ένας αλγόριθμος να τρέξει κάτω από αυτήν την μέθοδο πρέπει να συντρέχουν δυο πολύ σημαντικές ιδιότητες: i) Όλες οι εργασίες που θα τρέξουν γνωρίζονται από πριν, και ii) Είναι δυνατό να βρεθούν οι εξαρτήσεις μεταξύ των εργασιών χωρίς πρώτα να εκτελεστούν. Αν πάρουμε για παράδειγμα το Delaunay refinement που αναλύθηκε προηγουμένως, θα δούμε ότι κατά την διάρκεια της εκτέλεσης είναι πιθανό να μπούνε νέες εργασίες στην worklist (΄΄κακά΄΄ τρίγωνα) που σημαίνει ότι δεν ξέρουμε εκ των προτέρων τις εργασίες.

Σε αυτήν την μέθοδο όμως μπορεί να υπάρξει η εξαίρεση του *bulk-synchronous parallelism* [11]. Σε αυτήν την επέκταση χωρίζουμε την εκτέλεση σε βήματα. Σε κάθε βήμα εκτελείται ο αλγόριθμος *inspector-executor,* βρίσκονται οι εργασίες που μπορούν να εκτελεστούν, και στην συνέχεια προσδιορίζεται ένα μέγιστο σύνολο εργασιών άνευ εξαρτήσεων και εκτελείται. Στο επόμενο βήμα, οι εργασίες που δεν εκτελέστηκαν, μαζί με τις νέες εργασίες που δημιουργήθηκαν αποτελούν και πάλι το σύνολο της δουλειάς και ο *inspector-executor* εφαρμόζεται ξανά μέχρι να μην υπάρχουν άλλες εργασίες.

Δυναμικές Μέθοδοι: Στις δυναμικές μεθόδους, το κύριο χαρακτηριστικό είναι ο αισιόδοξος παραλληλισμός (optimistic ή speculative parallelism). Οι παράλληλες εργασίες του αλγορίθμου εκτελούνται και αν δεν έχουν εξαρτήσεις μεταξύ τους τότε τα αποτελέσματά τους γίνονται διαθέσιμα. Αν οι μεταξύ τους εξαρτήσεις παραβιάζονται τότε οι εργασίες σταματάνε και τα μέχρι τότε αποτελέσματα τους ανακαλούνται. Στην συνέχεια η εκτέλεση τους ξεκινάει ξανά, είτε παράλληλα με άλλες εργασίες είτε σειριακά. Το αποτέλεσμα των εργασιών γίνεται διαθέσιμο όταν αυτές τελειώσουν την εκτέλεση τους.

Η λογική αυτή είναι πολύ κοντά με τον αλγόριθμο Tomasulo, όπου για να γίνει καλύτερη εκμετάλλευση του παραλληλισμού σε επίπεδο εντολών, κομμάτια κώδικα εκτελούνται ταυτόχρονα και μόνο ένα από τα δύο γίνεται διαθέσιμο [12]. Αυτή η λογική μπορεί να υλοποιηθεί σε υψηλότερο επίπεδο μέσω *while-loops* στις διαθέσιμες εργασίες. Αν ο αριθμός των επαναλήψεων είναι γνωστός εκ των προτέρων, τότε η τεχνική αυτή (coarse-grain loop-level speculation), που έχει σε πολλά συστήματα υποστήριξη από το υλικό, ονομάζεται *thread-level speculation* [13] [14]. Αυτή η τεχνική, αν και προσφέρει πολλές από τις ιδιότητες που χρειάζονται οι irregular αλγόριθμοι, έχει δύο προβλήματα: i) Πρέπει να είναι γνωστός ο αριθμός των επαναλήψεων. Όπως είδαμε στο Delaunay refinement, αν η επανάληψη είναι πάνω στα ΄΄κακά΄΄ τρίγωνα στην worklist, τότε το άνω όριο της επανάληψης δεν είναι γνωστό καθώς νέα τρίγωνα εισάγονται στην worklist κατά την διάρκεια της εκτέλεσης. ii) Η τεχνική του thread level speculation βρίσκει τις εξαρτήσεις σε επίπεδο εντολών μηχανής για όλη την επανάληψη. Αυτό σημαίνει ότι αν υπάρχει μια παραβίαση read-write σε μια θέση μνήμης από δύο νήματα της επανάληψης τότε μια από αυτές τις εργασίες θα σταματήσει.

Για τις δομές δεδομένων που χρησιμοποιούν οι irregular αλγόριθμοι η δεύτερη συνθήκη είναι πολύ αυστηρή διότι ο παραλληλισμός τους προκύπτει από τις προσβάσεις στα δεδομένα. Για παράδειγμα, και μόνο η εισαγωγή ή διαγραφή εργασιών (΄΄κακών΄΄ τριγώνων στον Delaunay refinement) από την worklist θα μπορούσε να κάνει δύο νήματα να έχουν μια read write εξάρτηση.

Ένας ακόμη περιορισμός απόδοσης, είναι ότι σε thread-level speculation, η σειρά εκτέλεσης των επαναλήψεων είναι συγκεκριμένη, ενώ για παράδειγμα στον Delaunay refinement αλγόριθμο δεν έχει σημασία η σειρά επεξεργασίας των ΄΄κακών΄΄ τρίγωνών, παρά μόνο ο τελικός γράφος να είναι απαλλαγμένος από αυτά.

Τέλος, σε όλες αυτές τις τεχνικές δυναμικής εκτέλεσης, ένα σημαντικό κομμάτι του αλγορίθμου είναι ο συγχρονισμός. Μια fine-grain υλοποίηση ή ακόμα μια lock free, θα ήταν πολύ δύσκολο να υλοποιηθεί σωστά σε μια δομή δεδομένων όπως οι γράφοι. Αντιθέτως έχει προταθεί η λογική των transactions [15]. Σε αυτή την κατεύθυνση υπάρχουν αρκετές υλοποιήσεις σε Transactional Memory, software ή hardware. Το Transactional Memory θα συζητηθεί εκτενώς στην επόμενη ενότητα, καθώς θα είναι το κύριο χαρακτηριστικό σύγκρισης της απόδοσης των αλγορίθμων σε σχέση με το σύστημα Galois.

3 Transactional Memory

Η τεχνική του Transactional Memory για συγχρονισμό έχει ως στόχο να κάνει πιο εύκολη την συγγραφή παράλληλων προγραμμάτων, πετώντας το πρόβλημα των εξαρτήσεων και των deadlock, κρατώντας όμως την υπόσχεση για απόδοση συγκρίσιμη με fine-grain και lock free τεχνικές. Εν συντομία, οι επιλογές που έχει ένα προγραμματιστής όταν χρειάζεται να υλοποιήσει το σχήμα συγχρονισμού είναι οι παρακάτω:

- Coarse-grain: Το πρόβλημα του συγχρονισμού χωρίζεται σε ΄΄μεγάλες΄΄ εργασίες. Οι εργασίες αυτές εκτελούνται σειριακά στον επεξεργαστή χωρίς όμως να αφήνουν περιθώρια παραλληλισμού μέσα στην εργασία αυτή. Το πλεονέκτημα αυτού του τρόπου είναι η ευκολία υλοποίησης του. Το μειονέκτημα του είναι ότι δεν δίνει ευκαιρίες παραλληλισμού.
- Fine-grain: Το πρόβλημα του συγχρονισμού χωρίζεται σε πολλές μικρές εργασίες. Σε αντίθεση με πριν, οι εργασίες αυτές δεν μπορούν εκτελεστούν με οποιαδήποτε σειρά και ταυτόχρονα να ικανοποιούνται οι ACID ιδιότητες. Οι fine-grain τεχνικές απαιτούν προγραμματιστικό κόπο για να υλοποιηθούν σωστά, αλλά συνήθως συνοδεύονται και με καλή απόδοση στον παραλληλισμό.
- Lock-free: Οι προηγούμενες τεχνικές χρησιμοποιούν κλειδώματα στον πυρήνα τους για να προστατεύσουν ένα κρίσιμο τμήμα κώδικα. Αντιθέτως οι lock-free ή αλλιώς non-blocking χρησιμοποιούν ατομικές εντολές όπως Compare-and-swap (CAS). Ο προγραμματιστικός κόπος εδώ είναι μεγαλύτερος από αυτόν σε μια fine-grain υλοποίηση, αλλά προσφέρει συνήθως καλύτερη απόδοση και wait-free εγγύηση.

Η τεχνική Transactional Memory ευελπιστεί να προσφέρει την απόδοση μιας lock-free τεχνικής, χωρίς να χρησιμοποιεί locks (εγγύηση για wait-free) έχοντας παράλληλα την προγραμματιστική ευκολία μιας coarse-grain υλοποίησης.

3.1 Transactional Memory ως μοντέλο

Το μοντέλο συγχρονισμού που υλοποιεί το Transactional Memory (TM) είναι το ίδιο με αυτό των κλειδωμάτων. Τα TM συστήματα μπορούν να δώσουν εγγυήσεις Ατομικότητας (Atomicity), Συνέπειας (Consistency) και Απομόνωσης (Isolation) καθώς επίσης και wait-free και deadlock-free εγγυήσεις. Το κύριο χαρακτηριστικό ενός TM συστήματος είναι ότι δεν χρησιμοποιούνται κλειδώματα. Αντιθέτως, στην θέση των εντολών `lock_acquire` και `lock_release`, έχουμε αντίστοιχες εντολές της μορφής `transaction_start` και `transaction_end`, που ορίζουν με τον ίδιο τρόπο την εκτέλεση ενός κρίσιμου τμήματος κώδικα.

Όταν εκτελείται ένα κρίσιμο τμήμα από τουλάχιστον ένα νήματα οι εξαρτήσεις που μπορούν να υπάρξουν είναι οι εξής:

- Γίνεται μια ανάγνωση σε μια θέση μνήμης, την οποία ένα άλλο νήμα γράφει
- Γίνεται μια εγγραφή σε μια θέση μνήμης, την οποία ένα άλλο νήμα γράφει ή διαβάζει

Συνεπώς, στον πυρήνα τους τα TM συστήματα έχουν πάντα έναν μηχανισμό παρακολούθησης των εγγραφών σε επίπεδο load, store εντολών. Η λογική με την οποία δουλεύει είναι η ίδια με αυτήν των transactions στις βάσεις δεδομένων, μόνο που εδώ η ΄΄βάση΄΄ βρίσκεται στην κοινή μνήμη των δυο νημάτων. Σε αυτήν την λογική, το Transactional Memory υλοποιεί ένα πρωτόκολλο αισιόδοξης εκτέλεσης. Ένα transaction θα ξεκινήσει πάντα την εκτέλεση του άσχετα με τον αν εκείνη την στιγμή και τα άλλα νήματα εκτελούν το κρίσιμο τμήμα. Αν κατά την διάρκεια της εκτέλεσης ενός transaction δεν ανιχνευθεί καμία από τις δυο εξαρτήσεις που αναφέρθηκαν πριν, τότε το transaction εκτελείται επιτυχώς, και οι αλλαγές του γίνονται ορατές και διαθέσιμες στα υπόλοιπα νήματα. Με αυτό τον τρόπο ένα transaction κάνει commit.

Στην περίπτωση που ανιχνευθούν εξαρτήσεις τότε το transaction κάνει abort. Σε όλα τα TM συστήματα, αυτό σημαίνει την απόρριψη όλων των μέχρι τότε αλλαγών, και την επιστροφή του program counter σε κάποια εντολή. Αυτή η διεύθυνση επιστροφής, και γενικά το πως χειρίζεται ένα πρόγραμμα το transaction abort είναι στην ευχέρεια του προγραμματιστή. Το σημαντικό στοιχείο που προσφέρει το Transactional Memory είναι ότι σε περίπτωση που ανιχνευθεί κάποια εξάρτηση, τότε τα αποτελέσματα απορρίπτονται, σαν να μην εκτελέστηκε ποτέ ο κώδικας που τα προκάλεσε.

Το Transactional Memory λοιπόν, προσφέρει Ατομικότητα καθώς θα γίνουν όλες οι εγγραφές στην μνήμη ή καμία. Προσφέρει Απομόνωση καθώς οι αλλαγές γίνονται ορατές μόνο μετά το commit του transaction. Προσφέρει επίσης και πλήρη εξάλειψη των deadlocks λόγω της απουσίας των locks και όλα αυτά με μια φαινομενικά εύκολη προγραμματιστική διεπαφή. Τα TM συστήματα μπορούν όμως να υποφέρουν από live-lock και resource starvation και αυτό γιατί μπορεί μικρά transactions να εκτελούνται συνέχεια και να προκαλούν transaction abort σε άλλα μεγαλύτερα.

3.2 Χαρακτηριστικά Transactional Memory

Οι βασικές διαφοροποιήσεις μεταξύ των TM συστημάτων είναι i) ο τρόπος με τον οποίο χειρίζονται τα μέχρι τη στιγμή της εκτέλεσης αποτελέσματα όπως επίσης και τα αρχικά δεδομένα, σε περίπτωση που χρειαστούν για το rollback, και ii) ο τρόπος με τον οποίο ανιχνεύονται οι συγκρούσεις μεταξύ των transactions [16].

Data Versioning: Αυτό το χαρακτηριστικό ορίζει τον τρόπο που χρησιμοποιεί το TM σύστημα για να κρατάει τα παλιά και καινούρια δεδομένα, και στις περισσότερες υλοποιήσεις χωρίζεται σε Eager versioning και Lazy versioning.

Eager Data Versioning: Η τεχνική αυτή καταγράφει τις αλλαγές των store εντολών κατευθείαν στην μνήμη και κρατάει τις παλιές τιμές σε μια δομή που μοιάζει με στοίβα, η οποία παίζει τον ρόλο του undo-log. Σε κάθε νέα store εντολή μέσα στο transaction, η παλιά τιμή εισάγεται στην στοίβα και η νέα τιμή καταχωρείται στην μνήμη. Αν για κάποιο λόγο έχουμε conflict με κάποιο άλλο transaction και στην συνέχεια αυτό προκαλέσει abort, τότε η στοίβα διαβάζεται, και οι τιμές στην μνήμη επιστρέφουν στις αρχικές σε αντίστροφη σειρά. Αν το transaction ολοκληρωθεί κανονικά, τότε η undo-log στοίβα καταστρέφεται.

Τα πλεονεκτήματα αυτής τη τεχνικής είναι ότι τα commit είναι γρήγορα γιατί δεν απαιτείται καμία δοσοληψία με την μνήμη, ενώ τα μειονεκτήματα είναι i) Τα writes κατά την διάρκεια του transaction είναι πιο ‘’βαριά’’ γιατί απαιτούν και την λειτουργία μνήμης (store) αλλά και την ενημέρωση της undo-log στοίβας, ii) το abort είναι ακριβό γιατί απαιτεί το διάβασμα της στοίβας και εισαγωγή όλων την τιμών στην μνήμη, iii) καθώς επίσης και ότι ένα μη αναμενόμενο σταμάτημα της εφαρμογής αφήνει την μνήμη σε ασυνεπή κατάσταση.

Lazy Data Versioning: Η τεχνική αυτή είναι η ακριβώς αντίθετη με την προηγούμενη. Οι νέες τιμές που είναι να γραφούν στην μνήμη τοποθετούνται σε έναν buffer και η μνήμη κρατάει για όλη την διάρκεια της εκτέλεσης του transaction τις αρχικές τιμές. Αν έχουμε conflict και transaction abort τότε καμία αλλαγή στην μνήμη δεν χρειάζεται, παρά μόνο να πεταχτεί ο buffer, και αν έχουμε επιτυχή ολοκλήρωση και commit τότε ο write buffer γράφεται στην μνήμη.

Τα πλεονεκτήματα αυτή της τεχνικής είναι τα γρήγορα aborts καθώς το μόνο που απαιτείται είναι να πεταχτεί ο buffer, το γεγονός ότι η μνήμη βρίσκεται πάντα σε συνεπή κατάσταση καθώς κατά την διάρκεια του transaction κρατάει τις αρχικές τιμές καθώς επίσης και ότι τα writes είναι ‘’γρήγορα’’ αφού απαιτούν μόνο γράψιμο στον buffer και όχι στην μνήμη. Τα μειονεκτήματα είναι ότι τα commits είναι αργά, καθώς απαιτούν διάβασμα του buffer και γράψιμο στην μνήμη.

Eager Data Versioning	Lazy Data Versioning
+ Γρήγορα commits	+ Γρήγορα aborts
- Βαριά stores	+ Ελαφριά stores
- Αργά aborts	+ Συνεπής μνήμη
- Ασυνεπής μνήμη	- Αργά commits

Conflict Detection: Το χαρακτηριστικό αυτό αναφέρεται στην συμπεριφορά του TM συστήματος για το πως ανιχνεύεται μια σύγκρουση δεδομένων μεταξύ δύο transactions και για το πως επιλύεται αυτή η σύγκρουση. Και στις δύο περιπτώσεις υπάρχει η έννοια των read/write sets, τα οποία κρατάνε τις διευθύνσεις στις οποίες έχουν γίνει αντίστοιχες λειτουργίες. Οι διαφορετικές προσεγγίσεις που υπάρχουν εδώ είναι οι Pessimistic detection και Optimistic detection.

Pessimistic Conflict Detection: Η απαισιόδοξη τεχνική προσπαθεί να ανιχνεύσει τις συγκρούσεις όσο το δυνατόν νωρίτερα, την στιγμή δηλαδή που γίνεται μια load/store λειτουργία, σαν να είναι δηλαδή σίγουρη η ύπαρξη μιας εξάρτησης. Αν υπάρχει σύγκρουση τότε το σύστημα αποφασίζει αν θα γίνει abort ή καθυστέρηση στην εκτέλεση μέχρι το άλλο transaction να τερματίσει. Το σύστημα επίσης αποφασίζει ποιο από τα δύο transactions θα έχει προτεραιότητα.

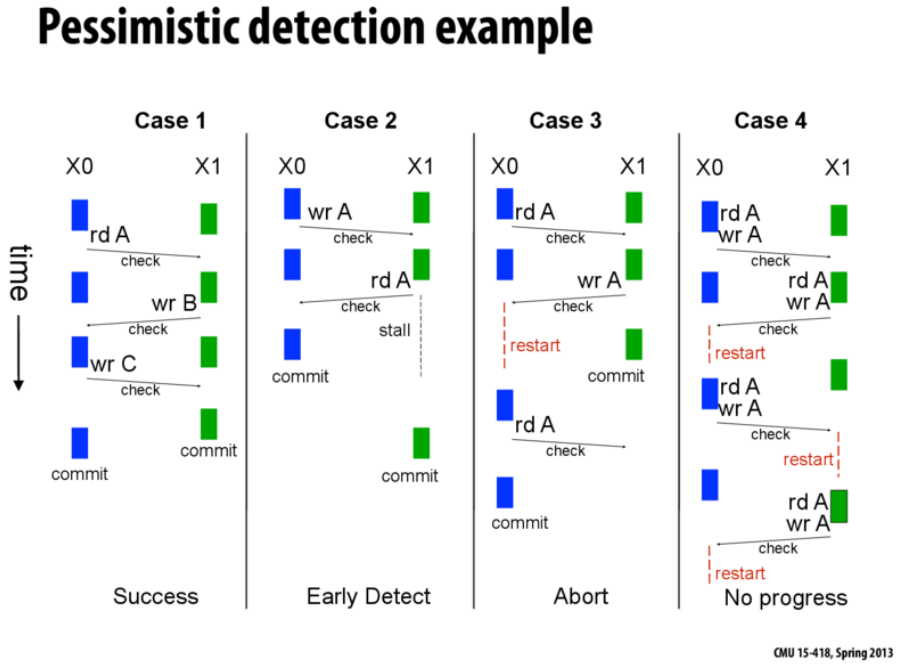

Figure 2: Pessimistic detection [17]

Στο παράδειγμα Figure 2 βλέπουμε ότι ο έλεγχος για σύγκρουση μεταξύ δύο transactions γίνεται αμέσως σε κάθε λειτουργία μνήμης. Στην δεύτερη περίπτωση, ο άμεσος έλεγχος προκαλεί την καθυστέρηση στο X1 μέχρι να τερματίσει την εκτέλεση το X0. Στην τρίτη περίπτωση όμως η εγγραφή του X1 προκαλεί abort στο X0. Στην συνέχεια το transaction προσπαθεί ξανά και επιτυγχάνει. Στην τέταρτη περίπτωση οι συνεχείς επανεκκινήσεις των transaction και ο άμεσος έλεγχος για σύγκρουση προκαλεί συνεχή abort χωρίς πρόοδο.

Το πλεονέκτημα της τεχνικής αυτής είναι ότι ανιχνεύει νωρίς τα conflicts και τα transactions δεν χάνουν χρόνο σε άσκοπη εκτέλεση, ενώ το μειονέκτημα είναι ότι υπάρχει η περίπτωση live-lock.

Optimistic Conflict Detection: Η αισιόδοξη τεχνική θεωρεί ότι το transaction δεν έχει εξαρτήσεις με άλλες δοσοληψίες και συνεχίζει την εκτέλεση του μέχρι το commit στάδιο. Σε εκείνο το σημείο το write-set που έχει κρατήσει ελέγχεται για συγκρούσεις με τα υπόλοιπα transactions που τρέχουν.

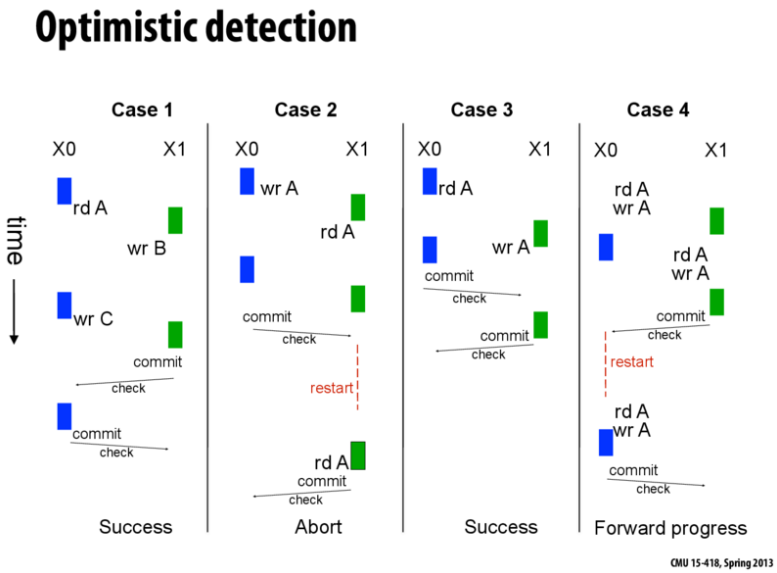

Figure 3: Optimistic Detection [17]

Στην πρώτη περίπτωση η Optimistic τεχνική θα τρέξει γρηγορότερα το transaction διότι θα γίνουν λιγότεροι έλεγχοι. Στην δεύτερη περίπτωση όμως, το commit του X0 προκαλεί το abort στο X1 καθώς δεν υπάρχει τρόπος να γίνει stall. Στην προηγούμενη περίπτωση το TM σύστημα μπορούσε να προκαλέσει καθυστέρηση, εδώ όμως δεν μπορεί να σταματήσει την εκτέλεση του X1. Στην τρίτη περίπτωση, επειδή οι συγκρούσεις ανιχνεύονται στο στάδιο του commit, τα transaction εκτελούνται κανονικά, σε αντίθεση με το Pessimistic πρωτόκολλο που προκαλούσε abort. Με την Optimistic τεχνική έχουμε περισσότερα δυνατά σειριοποιήσιμα μονοπάτια. Τέλος στην τέταρτη περίπτωση, το γεγονός ότι ο έλεγχος γίνεται στο τέλος δίνει λύση στο πρόβλημα του live-lock.

Τα πλεονεκτήματα της τεχνικής αυτής είναι ότι εγγυάται την πρόοδο σε κάθε περίπτωση. Τα μειονεκτήματα της είναι ότι δεν μπορεί να σταματήσει την άσκοπη δουλειά σε transactions που είναι καταδικασμένα να κάνουν abort (δεύτερη περίπτωση). Επίσης ένα πρόβλημα που προκύπτει είναι το starvation των transactions. Αυτό συμβαίνει στην τέταρτη περίπτωση για παράδειγμα όπου αν η X0 είναι μια μεγάλη και χρονοβόρα δοσοληψία τότε πολλές μικρές δοσοληψίες σαν την X1 μπορούν να προκαλούν συνεχή abort στην X0.

Contention Manager: Ένα κομμάτι του TM συστήματος είναι αυτός που παίρνει τις αποφάσεις για το ποια δοσοληψία θα κάνει abort και ποια θα συνεχίσει ή για το αν θα γίνει abort ή καθυστέρηση. Την δουλειά αυτήν την εκτελεί ο Contention Manager.

Υπάρχουν αρκετές πολιτικές που μπορούν να εφαρμοστούν όπως η Επιθετική πολιτική όπου ένα transaction κάνει abort κατευθείαν. Υπάρχει η Randomized πολιτική όπου το transaction που κάνει abort επιλέγεται τυχαία (λύνοντας το πρόβλημα του starvation) ή άλλες πολιτικές που μετράνε τον χρόνο που έχει αφιερωθεί στην εκτέλεση για κάθε transaction.

Επίσης, εκτός από την επιλογή του abort υπάρχει και η καθυστέρηση όπως φαίνεται και στο δεύτερο παράδειγμα.

Isolation: Τα TM συστήματα εγγυόνται απομόνωση στην εκτέλεση των transaction. Αυτό σημαίνει ότι Load εντολές δεν βλέπουν αποτελέσματα από Store εντολές που εκτελούνται μέσα σε ένα transaction και Store εντολές εκτός transaction προκαλούν εξαρτήσεις σε οποιαδήποτε transaction με Load εντολές στην συγκεκριμένη διεύθυνση. Αυτό το μοντέλο είναι Strong isolation. Αν κάποιο από τα παραπάνω δεν ικανοποιείται τότε έχουμε weak isolation.

Granularity: Το granularity ενός TM συστήματος αναφέρεται στο πόσο μακριά (απόσταση διευθύνσεων σε bytes) πρέπει να είναι Load/Store εντολές σε διαφορετικές

διευθύνσεις από δύο διαφορετικά transaction, χωρίς να προκαλούν conflict. Προφανώς, αν είναι στην ίδια διεύθυνση υπάρχει εξάρτηση. Ωστόσο, λόγω περιορισμών υλοποίησης, δεν είναι δυνατή η παρακολούθηση όλων των διευθύνσεων. Συνεπώς Load/Store εντολές σε κοντινές διευθύνσεις είναι πιθανό να προκαλέσουν Conflict. Το granularity ενός TM συστήματος είναι καθαρά θέμα υλοποίησης.

3.3 Υλοποιήσεις Transactional Memory

Συστήματα Transactional Memory έχουν υλοποιηθεί σε ένα μεγάλο αριθμό γλωσσών. Ωστόσο, για την πλήρη εκμετάλλευση των πλεονεκτημάτων τους, απαιτείται, τις περισσότερες φορές, υποστήριξη από το υλικό.

3.3.1 Software Transactional Memory (STM)

Το προγραμματιστικό μοντέλο ενός Software TM συστήματος έχει πολλές ομοιότητες με άλλες μεθόδους συγχρονισμού, όπως τα κλειδώματα. Ο προγραμματιστής απλά ορίζει την περιοχή του κώδικα την οποία επιθυμεί να εκτελεστεί ατομικά χρησιμοποιώντας ένα key word:

```
atomic {
      newNode->prev = node;
      newNode->next = node->next;
      node->next->prev = newNode;
      node->next = newNode;
}
```

Το σύστημα αναλαμβάνει την ευθύνη του συγχρονισμού έχοντας ως μόνη υποχρέωση την Απομόνωση. Ο προγραμματιστής πρέπει να φροντίσει για να παράγονται σωστά σειριοποιήσιμα προγράμματα και όχι το STM σύστημα. Ο τρόπος υλοποίησης, σε πολλές περιπτώσεις, διαφέρει από τα χαρακτηριστικά που αναφέρθηκαν πριν και περιλαμβάνει ένα μεγάλο εύρος τεχνικών που μπορεί να περιλαμβάνει και κλασσικές τεχνικές κλειδώματος. Κάποια STM συστήματα είναι τα παρακάτω:

- Η TinySTM [18], είναι μια βιβλιοθήκη υλοποιημένη σε C++ που μπορεί να χρησιμοποιηθεί σε 32 και 64 bit αρχιτεκτονικές σε Unix, Windows ή Mac OS, και χρησιμοποιεί ατομικές εντολές.
- Ο Intel STM compiler μπορεί να υποστηρίξει STM παράγοντας κώδικά για 32 και 64 bit αρχιτεκτονικές σε Intel ή AMD επεξεργαστές για Linux και Windows. Χρησιμοποιεί ατομικές εντολές και η χρήση του compiler αυτού είναι καθαρά για μεγάλης κλίμακας πειραματική-ερευνητική χρήση STM σε C++ προγράμματα
- Ο G++ 4.7 compiler παρέχει STM υποστήριξη ως ένα χαρακτηριστικό που βρίσκεται σε πειραματικό στάδιο
- Άλλες C++ υλοποιήσεις σε επίπεδο βιβλιοθήκης είναι οι LibLTX [19], LibCMT [20], TL2 [21], RSTM [22]
- Οι υλοποιήσεις σε Java περιλαμβάνουν την AtomJava, Deduce, ScalaStm και άλλες
- Κάποιες C# υλοποιήσεις είναι οι LibCMT, STMNet, NSTM. Οι δύο τελευταίες είναι καθαρές C# υλοποιήσεις για .NET προγραμματισμό.
- Το Mnesia υλοποιήθηκε από την Ericsson και είναι μια πραγματικού χρόνου-κατανεμημένο-υψηλής διαθεσιμότητας σύστημα που χρησιμοποιήθηκε στις τηλεπικοινωνίες. Το Mnesia είναι το κομμάτι της Erlang που υλοποιεί το STM σύστημα και είναι η παλαιότερη υλοποίηση ενός STM μοντέλου.
- Η DSTM [20] υλοποιεί το STM σύστημα στην Haskell
- Η AtomizeJS αποτελεί σύστημα STM για την Javascript και για NodeJS server
- Η coThreads είναι βιβλιοθήκη για την OCaml

- Άλλες πολλές υλοποιήσεις υπάρχουν επίσης και για Perl, Python, Ruby, Scala, Smalltalk.

3.3.2 Hardware Transactional Memory (HTM)

Υπάρχουν πολλά υπολογιστικά συστήματα τα οποία υποστηρίζουν την τεχνική του Transactional Memory στο υλικό. Η υλοποίηση αυτή είναι πάντα στο επίπεδο των πυρήνων του συστήματος, και στα κοινά δεδομένα που βλέπουν αυτοί, με την υποστήριξη μηχανισμών που υπάρχουν ήδη για Cache coherence (MESI). Ακόμη, αρκετές αλλαγές πρέπει να γίνουν ώστε για την υποθετική εκτέλεση που κάνουν τα transaction, να έχουν χώρο για να κρατάνε τις προσωρινές τιμές μέχρι να γίνει commit.

Μια αρκετά διάσημη τεχνική είναι να προθέτονται δυο έξτρα bits στην cache line (read, write bits) για να γίνεται η αναγνώριση των εξαρτήσεων μεταξύ των transactions. Όταν οι cache lines διαβάζονται ή γράφονται από transactions, τότε τα δύο έξτρα bits χρησιμοποιούνται. Βλέπουμε λοιπόν εδώ, ότι το granularity σε μια HTM υλοποίηση είναι το μέγεθος της cache line το οποίο μπορεί να προκαλεί λανθασμένες εξαρτήσεις.

Eager-Pessimistic: Σε αυτό το μοντέλο, όπου οι αλλαγές γίνονται κατευθείαν στην μνήμη και κρατούνται οι παλιές τιμές σε buffers, οι μεταβάσεις στο MESI κατά το διάβασμα μιας cache line δεν αλλάζουν. Όταν υπάρχει μια εντολή LOAD, και διαβάζεται μια cache line γίνονται οι τυπικές μεταβάσεις (S -> S, I -> S ή I->E) και χρησιμοποιείται το R bit. Όταν υπάρχει μια STORE εντολή και γράφεται μια cache line γίνονται οι μεταβάσεις (S->M, E->I, I-M), και χρησιμοποιείται το W bit. Ο undo-log buffer κρατάει την προηγούμενη τιμή της cache line σε περίπτωση abort.

Για να ανιχνευθεί η σύγκρουση δεδομένων χρησιμοποιείται και πάλι το MESI πρωτόκολλο. Όταν μια cache δει ένα load request από μια άλλη cache λόγω read miss, και έχει την γραμμή που ζητείται, τότε αν δεν είναι σε transaction η συγκεκριμένη γραμμή ή έχει το R bit μόνο, την παρέχει. Αν όμως η γραμμή έχει το W bit, τότε έχουμε σύγκρουση και το TM σύστημα θα αποφασίσει για την τύχη των δύο transaction. Παρομοίως, όταν μια cache θέλει να αλλάξει την κατάσταση μιας γραμμής σε M λόγω STORE εντολής, το exclusive load request θα προκαλέσει τις άλλες caches να κοιτάξουν τις γραμμές τους, και αν την έχουν είτε σε R είτε σε W υπάρχει conflict.

Το commit δεν απαιτεί κάποια ιδιαίτερη πράξη καθώς οι αλλαγές γίνονται κατευθείαν στην μνήμη. Απαιτείται μόνο το καθάρισμα των R,W bits καθώς οι cache lines δε είναι πια σε εκτέλεση μέσα σε transaction.

Το abort είναι ακριβό, καθώς οι αρχικές cache lines στον buffer πρέπει να διαβαστούν και γραφτούν ξανά. Αυτή η διαδικασία έχει αρκετό κόστος καθώς άλλα reads, writes θα πρέπει να περιμένουν μέχρι η σωστή έκδοση της κάθε cache line να μεταφερθεί στην αρχική κατάσταση.

Lazy-Optimistic: Μια τεχνική που κρατάει την μνήμη σε αρχική κατάσταση πάντα, τις αλλαγές σε έναν buffer και ελέγχει για conflict στο commit.

Το διάβασμα μιας cache line είναι παρόμοιο με αυτό ενός Eager-pessimistic συστήματος. Όταν έχουμε γράψιμο σε μια cache line, τα δεδομένα γράφονται και πάλι στην cache κανονικά και κρατούνται οι παλιές cache lines στο write buffer με τα άλλα νήματα να έχουν πρόσβαση στα παλιά δεδομένα. Η υλοποίηση αυτής τεχνικής μπορεί να προκαλέσει αρκετά προβλήματα υπερχείλισης της cache. Το γεγονός όμως ότι δεν γίνεται επικοινωνία κατά την διάρκεια ενός transaction για έλεγχο εξαρτήσεων αλλά μόνο στο commit κάνει αυτό το μοντέλο βιώσιμο.

Το commit όμως απαιτεί δουλειά. Ο επεξεργαστής στέλνει τις διευθύνσεις των δεδομένων που διαβάστηκαν στο δίκτυο και αν ανιχνευθεί εξάρτηση μεταξύ των caches τότε το TM σύστημα επιλύει την σύγκρουση. Ο έλεγχος αυτός γίνεται από τις caches με τον ίδιο τρόπο και με την Eager-Pessimistic, κοιτώντας τα R, W bits των cache lines. Αν δεν ανιχνευθεί εξάρτηση, τότε καθαρίζονται τα R, W bits των cache lines του transaction, και οι άλλες caches

ξέρουν ότι έχουν dirty δεδομένα (γιατί διαβάζανε τα παλιά), λόγω του πακέτου που στάλθηκε πριν.

Το Abort είναι εύκολο καθώς κρατιόνται στην μνήμη και οι δύο εκδόσεις οπότε σβήνονται τα R, W bits των γραμμών αυτών και γίνονται Invalid και κρατούνται οι αρχικές cache lines.

Lazy-Pessimistic: Η τεχνική αυτή συνδυάζει χαρακτηριστικά των δυο προηγούμενων όπου οι αλλαγές κρατούνται σε write-buffer και ο έλεγχος εξαρτήσεων γίνεται σε κάθε εντολή. Χρησιμοποιεί την Lazy τεχνική για να κρατάει και τα παλιά δεδομένα, και χρησιμοποιεί το cache coherence protocol και ανίχνευση συγκρούσεων.

Το commit είναι εύκολο καθώς το pessimistic πρωτόκολλο ελέγχει συνεχώς για συγκρούσεις με τα άλλα transactions. Ωστόσο, τα δεδομένα πρέπει να μεταφερθούν από τον write buffer στην μνήμη.

Το abort είναι ίδιο με την Lazy-Optimistic.

Eager-Optimistic: Η τεχνική αυτή δεν τυγχάνει ευρείας χρήσης καθώς το Eager πρωτόκολλο γράφει τις αλλαγές στην cache κατευθείαν, προκαλώντας τα cache coherence μηνύματα, αλλά το Optimistic πρωτόκολλο περιμένει μέχρι το commit για να βρει εξαρτήσεις ξοδεύοντας και χρόνο και πόρους στην μνήμη αφήνοντας transactions να τρέχουν όπου θα μπορούσε να είχε βρεθεί εξάρτηση.

Αντιθέτως η τεχνική αυτή έχει υλοποιήσεις σε STM συστήματα όπως τα Batrok-STM [23] και McRT [24]. Μια Lazy υλοποίηση σε ένα STM σύστημα πρέπει να διαβάζει τον write-buffer συχνά και στο τέλος να γράψει και όλες τις τιμές του στην μνήμη κάτι που είναι ακριβό καθώς δεν είναι η cache δομή πια που μπορεί να κρατάει αυτές τις τιμές αλλά μια high level δομή δεδομένων στην μνήμη του προγράμματος. Άρα στα STM προτιμάται το Eager πρωτόκολλο. Επίσης ο έλεγχος για conflicts, κάθε φορά που έχουμε εγγραφή, στις high level δομές που κρατάνε τις τιμές μπορεί επίσης να είναι ακριβός, καθώς δεν υπάρχει η ταχύτητα της cache και το MESI πρωτόκολλο, ενώ η optimistic τεχνική που ελέγχει μια φορά στο commit του transaction έχει τα πλεονεκτήματα του bulk-processing.

Transactional Memory implementations		
Conflict\Version	Lazy	Eager
Optimistic	TCC [25], FlexTM [26], BulkTM [27]	Batrok-STM, McRT
Pessimistic	LTM [28], VTM [29]	LogTM [30], UTM [28], MetaTM [31]

3.3.3 Hybrid Transactional Memory

Η τεχνική αυτή αποτελεί μια μίξη των προηγούμενων, όπου έχεις STM συστήματα να χρησιμοποιούν εντολές που προφέρουν υποστήριξη από το υλικό για επιτάχυνση. Ορισμένες υλοποιήσεις αυτών είναι το PHyTM [32], Invyswell [33].

3.4 Υποστήριξη επεξεργαστών

Ένας από τους πρώτους επεξεργαστές που ενσωμάτωσε την τεχνολογία του Transactional Memory σαν μια μορφή υποθετικής εκτέλεσης ενός νήματος ήταν ο Rock επεξεργαστής από την Sun Microsystems το 2009 [34]. Αν και η τεχνολογία απέδειξε ότι μπορούσε να χρησιμοποιηθεί στην πράξη, κυρίως σε Hybrid υλοποιήσεις, ο επεξεργαστής δεν βγήκε ποτέ στην αγορά. Ορισμένα εργοστασιακά μοντέλα διατέθηκαν σε ερευνητικά κέντρα για ακαδημαϊκή έρευνα πάνω στην τεχνολογία.

Το 2009 η AMD πρότεινε μια οικογένεια από x86 εντολές, την Advanced Synchronization Facility, ως μια μορφή περιορισμένης υποστήριξης Transactional Memory. Η λογική ήταν να παρέχονται hardware εντολές που θα μπορούσαν να χρησιμοποιηθούν σε υψηλού επιπέδου υλοποιήσεις συστημάτων συγχρονισμού όπως TM και lock-free σχήματα. Μέχρι και τον Οκτώβριο του 2013 το συγκεκριμένο σετ εντολών ήταν στην φάση της αξιολόγησης.

Το 2011 η IBM ανακοίνωσε ότι το Blue Gene/Q είχε υποστήριξή από το υλικό για το SpTM. Η τεχνολογία αυτή είχε δύο λειτουργίες. Στην πρώτη ένα write από ένα transaction μπορεί να προκαλέσει σύγκρουση με ένα transaction που διαβάζει την ίδια θέση μνήμης. Η δεύτερη λειτουργία είναι η δυνατότητα διαφορετικά νήματα να έχουν διαφορετικές εκδόσεις της ίδιας θέσης μνήμης και το hardware να θυμάται την σειρά και την ‘’ηλικία’’ των νημάτων για να υλοποιεί τον συγχρονισμό. Τα ‘’νεότερα’’ μόνο μπορούν να διαβάζουν δεδομένα από τα ‘’παλιότερα’’ νήματα, και οι εγγραφές γίνονται με βάση την σειρά των νημάτων.

Η πιο πρόσφατη υλοποίηση είναι αυτή από την Intel στις Haswell, Broadwell, Skylake και στην τελευταία Kaby Lake αρχιτεκτονική. Η Intel περιγράφει το Transactional Memory μόνο από την πλευρά του χρήστη, και πως αυτός μπορεί να το χρησιμοποιήσει, χωρίς να δίνει τεχνικές λεπτομέρειες για την υλοποίηση του. Στα παρακάτω κεφάλαια θα αναλυθούν οι λεπτομέρειες της τεχνολογίας αυτής, και θα μετρηθεί η απόδοση της σε εφαρμογές που απαιτούν συγχρονισμό.

3.5 Intel Transactional Memory

Στόχος της Intel με την υλοποίηση ενός Transactional Memory συστήματος είναι να βελτιωθεί η απόδοση των προγραμμάτων που χρησιμοποιούν τεχνικές συγχρονισμού βασισμένες σε κλειδώματα και ταυτόχρονα να παραμείνει ίδιο το μοντέλο προγραμματισμού. Για αυτό τον σκοπό έχει υλοποιήσει ένα νέο σετ εντολών το οποίο ονομάζει Transactional Synchronization Extensions (TSX). Το TSX επιτρέπει στον επεξεργαστή να προσδιορίζει δυναμικά εάν τα νήματα χρειάζεται να σειριοποιηθούν λόγω μιας περιοχής κώδικας προστατευμένης από ένα lock και να εφαρμόζουν την σειριοποίηση μόνο εάν χρειάζεται. Αυτή η τεχνική επιτρέπει στο υλικό να εκμεταλλεύεται κρυμμένο παραλληλισμό λόγω αχρείαστου συγχρονισμού. Αυτή η τεχνική ονομάζεται από την Intel, Hardware Lock Elision (HLE) και είναι για εύκολη μετατροπή legacy κώδικα που χρησιμοποιεί κλειδώματα σε transactional εκτέλεση.

Η Intel προσφέρει επίσης μια δεύτερη τεχνική, την οποία ονομάζει Restricted Transactional Memory (RTM), η οποία επιτρέπει στον προγραμματιστή να ορίζει περιοχές κώδικα για συγχρονισμό, με ειδικές εντολές, και να ορίζει επίσης και μια εναλλακτική εκτέλεση σε περίπτωση που η εκτέλεση του transaction αποτύχει. Αυτή η τεχνική είναι το κλασσικό Transactional Memory που συζητήθηκε στις προηγούμενες ενότητες.

Αν ένα transaction ολοκληρωθεί επιτυχημένα, τότε το hardware εγγυάται ότι οι αλλαγές που γίναν μέσα στο transaction θα είναι γίνουν ορατές όλες μαζί τους υπόλοιπους λογικούς πυρήνες. Ένας επεξεργαστής κάνει ορατές τις αλλαγές στους άλλους επεξεργαστές μόνο στο commit, μια τεχνική που η Intel ονομάζει Atomic commit χωρίς όμως να προσδιορίζεται αν πρόκειται για Eager ή Lazy versioning.

Αν ένα transaction αποτύχει, τότε ο πυρήνας θα επαναφέρει την εκτέλεση αναιρώντας όλες τις αλλαγές με μια τεχνική που η Intel ονομάζει transactional abort. Το hardware θα φροντίσει ώστε να φανεί σαν να μην έγινε ποτέ η εκτέλεση του συγκεκριμένου κώδικα, και θα συνεχίσει την εκτέλεση σύμφωνα με τον κώδικα του προγραμματιστή.

Η Intel υλοποιεί την μέθοδο των read, write sets που εξηγήθηκε προηγουμένως, χωρίς όμως να αναφέρει αν χρησιμοποιεί Optimistic, ή Pessimistic detection. Αναφέρεται όμως ότι

ο έλεγχος για σύγκρουση γίνεται στο επίπεδο της L1 cache line μέσω του cache coherence protocol, που σημαίνει 64 bytes granularity.

Το TSX αποτελεί σύνολο εντολών σε C, C++ κώδικα. Το κεφάλαιο αυτό είναι γραμμένο σύμφωνα με τα Reference Documentation που παρέχονται από την Intel [35], [36].

3.5.1 Hardware Lock Elision

Η τεχνική αυτή αποτελεί μέθοδο συγχρονισμού για υποστήριξη παλαιότερων προγραμμάτων, που δεν έχουν γραφτεί για να κάνουν χρήση του RTM και χρησιμοποιούν κλειδώματα. Σε μια τέτοια εκτέλεση το κλείδωμα διαβάζεται στην αρχή του transaction, χωρίς να γράφεται, με την πρόβλεψη ότι η τιμή του δεν θα αλλάξει κατά την διάρκεια της εκτέλεσης. Οι δύο εντολές που χρησιμοποιούνται σε αυτήν την περίπτωση είναι οι XACQUIRE και XRELEASE.

Ο προγραμματιστής αλλάζει την εντολή που έχει για να παίρνει το lock με την XACQUIRE, και ο επεξεργαστής υλοποιεί τον συγχρονισμό. Αν και το lock acquire αποτελεί εγγραφή σαν εντολή, ο επεξεργαστής δεν το βάζει στο write set, παρά μόνο στο read set. Με τον τρόπο αυτό, αν το lock ήταν διαθέσιμο πριν το ξεκίνημα του transaction, οι υπόλοιποι επεξεργαστές θα συνεχίσουν να το βλέπουν διαθέσιμο και θα μπορούν να μπουν και να εκτελέσουν το κρίσιμο τμήμα. Αν στην πορεία προκύψει σύγκρουση άλλων δεδομένων τότε το σύστημα θα την χειριστεί κατάλληλα.

Στο σημείο του κώδικα που αφήνεται το lock, ο προγραμματιστής χρησιμοποιεί την XRELEASE εντολή. Αν η εντολή αυτή σχετίζεται με την αντίστοιχη XACQUIRE εντολή τότε ο επεξεργαστής αποφεύγει το γράψιμο στο lock και κάνει commit το transaction.

Με τον τρόπο αυτό, αν δύο νήματα εκτελούν ένα κρίσιμο τμήμα κώδικα προστατευμένο από το ίδιο lock, αλλά χωρίς να έχουν κάποια εξάρτηση δεδομένων, τότε η εκτέλεση τους θα γίνει ταυτόχρονα, αποφεύγοντας έτσι το coarse grain σχήμα. Αν κάποιος όμως γράψει στο lock non-transactionally, το γεγονός ότι βρίσκεται στο read-set των άλλων transaction θα προστατέψει την εκτέλεση τους με abort.

3.5.2 Restricted Transactional Memory

Η τεχνική αυτή αποτελεί το μοντέλο προγραμματισμού που περιεγράφηκε θεωρητικά προηγουμένως. Η Intel δίνει συγκεκριμένες εντολές στον προγραμματιστή με τις οποίες ορίζει transactional περιοχές. Το καινούριο χαρακτηριστικό εδώ είναι το γεγονός ότι ο προγραμματιστής πρέπει να ορίσει ένα μονοπάτι εκτέλεσης σε περίπτωση abort. Οι εντολές που είναι διαθέσιμες σε C,C++ είναι οι παρακάτω:

- *`unsigned int _xbegin(void)`*: Προσδιορίζει την αρχή ενός transactional region. Με την τιμή επιστροφής ελέγχεται αν ξεκίνησε το transaction με επιτυχία. Η εντολή αυτή αποτελεί και την διεύθυνση επιστροφής σε περίπτωση abort. Αν ένα transaction ξεκινήσει επιτυχώς τότε επιστρέφεται 0xFFFFFFFF, αν όμως επιστραφεί κάτι άλλο σημαίνει ότι έχουμε transaction abort. Σε εκείνο το σημείο ο προγραμματιστής ορίζει και το fallback path.
- *`unsigned int _xtest(void)`*: Με την εντολή αυτή ελέγχεται αν βρισκόμαστε σε transactional εκτέλεση.
- *`void _xend(void)`*: Με την εντολή αυτή προσδιορίζεται το τέλος ενός transaction και ο επεξεργαστής θα προσπαθήσει να κάνει commit. Αν αυτό αποτύχει τότε θα αναιρεθούν οι αλλαγές και η εκτέλεση θα επιστρέψει στο _xebgin() με έναν κωδικό λάθους.
- *`void _xabort(const unsigned int imm)`*: Προκαλεί abort σε ένα transaction και η εκτέλεση συνεχίζεται από το fallback path.

Το RTM χρησιμοποιεί τον EAX καταχωρητή καθόλη την διάρκεια του transaction, και μέσω αυτού επικοινωνεί την κατάσταση του transaction στο ανώτερο επίπεδο.

EAX register bit	Σημασία
0	1 αν έχουμε abort λόγω _xabort
1	Αν είναι 1 τότε το transaction είναι πιθανό να επιτύχει σε επόμενη εκτέλεση
2	1 αν έχουμε σύγκρουση δεδομένων με άλλη εκτέλεση
3	1 αν έχουμε υπερχείλιση μνήμης
4	1 αν έχουμε abort λόγω breakpoint debug
5	1 αν έχουμε φωλιασμένα transactions
23:6	Δεν δίνονται πληροφορίες
31:24	Το όρισμα της _xabort μόνο αν και το bit 0 είναι 1

Με τον τρόπο αυτό μπορεί ο προγραμματιστής να προσδιορίσει τον λόγο του abort.

3.5.3 Transactional Aborts

Τα Transactional aborts μπορεί να προκληθούν για πολλούς λόγους, και είναι το κύριο σημείο που πρέπει να γίνει βελτιστοποίηση. Η κλασσική περίπτωση για ένα abort είναι η σύγκρουση δεδομένων. Η Intel αναφέρει ότι αυτός ο έλεγχος γίνεται μέσω του cache coherence protocol και το transaction που ανιχνεύει την εξάρτηση θα κάνει και abort. Γενικά και από τον προηγούμενο, πίνακα τα aborts είναι τα παρακάτω:

- Συγκρούσεις δεδομένων
- Συγκρούσεις δεδομένων στην μεταβλητή κλειδώματος (HLE μόνο)
- Υπερχείλισή μνήμης
- Πειρασμούς της HLE εκτέλεσης
- Σαφή abort λόγω _xabort
- Φωλιασμένα transactions (το HLE υποστηρίζει 1, και το RTM έως 7)
- Debug breakpoints

Παρακάτω θα αναλυθούν οι δύο σημαντικότεροι λόγοι για abort.

3.5.3.1 Συγκρούσεις δεδομένων

Μια σύγκρουση δεδομένων προκαλείται είτε όταν μια δοσοληψία διαβάζει μια διεύθυνση η οποία είναι στο write-set μια άλλης δοσοληψίας, είτε όταν μια δοσοληψία γράφει μια διεύθυνση η οποία είναι στο read ή στο write set μιας άλλης δοσοληψίας. Η Intel αναφέρει επίσης ότι στην αρχική υλοποίηση η έλεγχος γίνεται μέσω του cache coherence protocol στο granularity μιας cache line.

False sharing: Συγκρούσεις που προκύπτουν λόγω read, write σε διαφορετικές διευθύνσεις στην ίδια cache line. Το παραγέμισμα των δομών δεδομένων ώστε μεταβλητές να βρεθούν σε διαφορετικές γραμμές είναι μια τεχνική ελαχιστοποίησης αυτού του παράγοντα.

True sharing: Συγκρούσεις λόγω read, write στην ίδια διεύθυνση

Statistics Maintenance: Αρκετές φορές μπορεί να χρησιμοποιηθούν τεχνικές για συλλογή στατιστικών οι οποίες να προκαλούν aborts. Πρέπει να δίνεται προσοχή μέσα στο transaction να είναι μόνο εντολές σχετικές με το κρίσιμο τμήμα. Η συλλογή στατιστικών ανά νήμα είναι μια καλή τεχνική για αποφυγή αυτών των aborts

Accounting Data Structures: Όταν το transaction έχει να κάνει ανανέωση μιας δομής δεδομένων, όπως μέτρηση εγγραφών ή αναδιοργάνωση δέντρων ή σωρών, τότε πρέπει να δίνεται προσοχή διότι μπορεί να είναι μεγάλα και χρονοβόρα transactions και να προκαλούν πολλά aborts όχι μόνο στον εαυτό τους αλλά και στα υπόλοιπα που τρέχουν εκείνη την στιγμή.

Conditional Writes: Μια εγγραφή σε μια μεταβλητή είναι write operation, άσχετα με το αν η προηγούμενη τιμή και η καινούρια είναι διαφορετικές. Αυτό μπορεί να προκαλέσει αχρείαστα abort.

Προκαλεί πάντα εγγραφή	Μπορεί να αποφευχθεί μερικές εγγραφές
state = true;	if (state != true) state = true;

3.5.3.2 Υπερχείλιση μνήμης

Οι περιορισμοί μνήμης που υπάρχουν στο hardware μπορούν να προκαλέσουν abort σε ένα transaction που χρησιμοποιεί πολύ μνήμη. Διαβάζοντας και γράφοντας πολλές διευθύνσεις μεγαλώνει το μέγεθος των read, write sets. Το hardware δεν προσφέρει εγγύηση για τους πόρους που θα διατεθούν ούτε ότι μια δοσοληψία σίγουρα κάποια στιγμή θα επιτύχει. Η Intel αναφέρει ότι ο επεξεργαστής κρατάει και read-set και write-set με τις διευθύνσεις στο πρώτο επίπεδο της cache.

Μια cache line που διώχνεται από την L1 cache μπορεί να μην προκαλέσει κατευθείαν abort καθώς η γραμμή αυτή μπορεί να παραμείνει στην L2. Στην Haswell αρχιτεκτονική η cache αυτή κρατάει την γραμμή πιθανολογικά καθώς το δεύτερο επίπεδο μπορεί να μην έχει την ίδια δομή με το πρώτο. Το τι γίνεται σε αυτή την περίπτωση είναι καθαρά θέμα υλοποίησης σε κάθε αρχιτεκτονική.

Στις Haswell, Broadwell και SkyLake αρχιτεκτονικές, η L1D cache έχει συσχετισμό (associativity) 8. Αυτό σημαίνει ότι 9 εγγραφές σε διακριτές περιοχές που θα έρθουν στην ίδια cache line θα προκαλέσουν abort. Ωστόσο, λόγω περιορισμών του υλικού, λιγότερες εγγραφές δεν σημαίνει ότι δεν θα συμβεί ποτέ abort. Επίσης, όταν δύο νήματα μοιράζονται τους πόρους του ίδιου πυρήνα λόγω HyperThreading, το μέγιστο επιτρεπτό μέγεθος των read/write sets μειώνεται σημαντικά .

Ακόμη, aborts μπορεί να προκληθούν όταν υπάρχουν κλήσεις σε συναρτήσεις βιβλιοθήκης ή σε system calls, όπου το μέγεθος του εκτελούμενου κώδικα αυξάνεται σημαντικά.

Οι περιορισμοί μνήμης, μπορεί να είναι ένας σημαντικός περιοριστικός παράγοντας, και θα συζητηθεί ξεχωριστά σε κάθε παράδειγμα αλγορίθμου.

3.5.4 Fallback μονοπάτι

Ένα σημαντικό κομμάτι είναι το πως θα χειριστεί ο κώδικας ένα transactional abort. Είναι σημαντικό διότι το RTM δεν δίνει καμία εγγύηση ότι κάποιο transaction κάποτε θα πετύχει και θα κάνει commit. Για αυτό το λόγο ο προγραμματιστής θα πρέπει να έχει προσφέρει ένα μονοπάτι εκτέλεσης που εγγυάται την πρόοδο της. Στην τεχνική του lock elision δεν γράφει ο προγραμματιστής fallback μονοπάτι, αλλά στην πραγματικότητα υπάρχει από το ίδιο το σύστημα. Είναι το lock acquire που γίνεται σε περίπτωση transaction abort, το οποίο θα προκαλέσει abort σε όποιο transaction έχει εκείνη την στιγμή το lock στο read set. Στην RTM το fallback μονοπάτι δίνεται στον κώδικα ανάλογα με την τιμή επιστροφής του *_xbegin()*.

Στο fallback μονοπάτι ο προγραμματιστής έχει το περιεχόμενο του EAX καταχωρητή που δίνει τον λόγο του abort. Σε αυτό το σημείο είναι δυνατή η καταγραφή στατιστικών και η αλλαγή της εκτέλεσης αν αυτό χρειάζεται. Αυτό που προτείνεται από την Intel είναι ο κώδικας να προσπαθεί ξανά να εκτελέσει το transaction, όχι όμως αμέσως, καθώς αν είχαμε abort λόγω data conflict τότε το να προσπαθήσεις ξανά να εκτελέσεις το transaction απλά φέρνεις την δοσοληψία στην ίδια κατάσταση με πριν. Αν όμως ήταν abort λόγω capacity, τότε οι δυνατότητες σου είναι περιορισμένες. Αν τα δεδομένα του transaction είναι πολλά και δεν χωράνε στην L1D cache τότε μπορεί να ευθύνεται η Hyper Threading τεχνολογία, αλλά μπορεί να ευθύνεται και ο κώδικας καθώς μεγάλα transactions απλά δεν μπορούν να εκτελεστούν.

Σαν γενική οδηγία η Intel προτείνει σε συστήματα με μεγάλο αριθμό από πυρήνες οι προσπάθειες για εκτέλεση του transaction να είναι πολλές.

Στην δική μου υλοποίηση, το transaction προσπαθεί συγκεκριμένο αριθμό φορών, ανάλογα με ένα argument, και αν αποτύχει σε όλες, τότε πηγαίνει σε ένα global lock και περιμένει να εισέλθει στο κρίσιμο τμήμα. Παρακάτω θα μελετηθεί πως ο αριθμός αυτός επηρεάζει την απόδοση.

3.5.5 Macros, συναρτήσεις και συλλογή στατιστικών

Για τον ευκολότερο προσδιορισμό των transactional regions, χρησιμοποιήθηκαν τα παρακάτω macros σε C, C++:

```
#define __try_transaction(x) if ((x = _xbegin()) == _XBEGIN_STARTED)
#define __transaction_abort(c) _xabort(c);
#define __transaction_end if (_xtest()) _xend();
```

Και ο κώδικας για τον ορισμό της αρχής και του τέλους ενός transaction είναι οι δύο παρακάτω συναρτήσεις:

```
static inline int transaction_start(pthread_spinlock_t * fallback_lock, int
number_of_tries){

#ifdef RTM_GATHER_STATS
        ReportStat& stats = ReportStat::getInstance();
#endif

       int status;
       int aborts=0;
       while(1){
              while(*fallback_lock == 0);

              __try_transaction(status){
                     if (*fallback_lock == 0) __transaction_abort(0xff);

                     return number_of_tries - aborts;
              }

#ifdef RTM_GATHER_STATS
              stats.report_status(status);
#endif
              if (aborts++ == number_of_tries){
                     pthread_spin_lock(fallback_lock);
                     return number_of_tries - aborts;
              }
       }
       return -1;
}

static inline int transaction_end(pthread_spinlock_t * fallback_lock){

#ifdef RTM_GATHER_STATS
       ReportStat& stats = ReportStat::getInstance();
#endif
       if (*fallback_lock == 1){
              __transaction_end;
#ifdef RTM_GATHER_STATS
              stats.report_transaction();
#endif
              return 0;
       }else{
```

```
            pthread_spin_unlock(fallback_lock);
            return 1;
      }

}
```

Με τις δύο αυτές συναρτήσεις, ο προγραμματιστής ορίζει το transactional region, δίνοντας ως όρισμα ένα global lock, και τον αριθμό των φορών που θα προσπαθήσει να κάνει το transaction. Σε περίπτωση transaction abort, τότε καταγράφει τα στατιστικά σε μια global δομή *ReportStat*. Η δομή-κλάση αυτή κρατάει τα στατιστικά ανά νήμα χρησιμοποιώντας την *sched_getcpu()* που επιστρέφει το id του λογικού πυρήνα στον οποίο εκτελείται ο κώδικας. Οι δύο συναρτήσεις που χρησιμοποιούνται για την καταγραφή είναι οι εξής:

```
void ReportStat::report_status(int status){
      int thread_id = sched_getcpu();

      stats[thread_id]->aborts++;
      if(status & _XABORT_CONFLICT){
            stats[thread_id]->abort_conflict += 1;
      }else if (status & _XABORT_CAPACITY){
            stats[thread_id]->abort_capacity += 1;
      }else if (status & _XABORT_RETRY){
            stats[thread_id]->abort_retry += 1;
      }else if (status & _XABORT_EXPLICIT){
            stats[thread_id]->abort_explicit += 1;
      }else if (status & _XABORT_DEBUG){
            stats[thread_id]->abort_debug += 1;
      }else if (status & _XABORT_NESTED){
            stats[thread_id]->abort_nested += 1;
      }
}

void ReportStat::report_transaction(){
      int thread_id = sched_getcpu();

      stats[thread_id]->transactions_completed += 1;
}
```

Οι δομές που χρησιμοποιούνται παραπάνω είναι οι εξής:

```
typedef struct per_thread_stats{
      int abort_conflict, abort_capacity, abort_retry, abort_explicit,
abort_debug, abort_nested, aborts, transactions_completed, locks;
      int dummy[7];
}per_thread_stats_t;
per_thread_stats_t ** stats;
```

Το struct που κρατάει για κάθε thread στατιστικά έχει padding 7 ακεραίων για να είναι ακριβώς 64 bytes δομή για κάθε νήμα, όσο και το μέγεθος μιας L1D cache line στους Intel επεξεργαστές.

Ο χρήστης έχει την υποχρέωση να ορίσει και να αρχικοποιήσει το *pthread_spinlock_t*, και να ορίσει την σημαία *RTM_GATHER_STATS* αν επιθυμεί την συλλογή στατιστικών για τα transactions.

4 Το σύστημα Galois

Το σύστημα Galois δημιουργήθηκε για την εκμετάλλευση παραλληλισμού irregular αλγορίθμων. Αποτελείται από προγραμματιστικές δομές που φαίνονται στον χρήστη μέσω βιβλιοθηκών για υλοποίηση προγραμμάτων και ένα runtime σύστημα. Το μοντέλο προγραμματισμού που χρησιμοποιεί το Galois είναι αυτό που συζητήθηκε και στην Ενότητα 2.2, δηλαδή worklists που κρατάνε τον αριθμό των εργασιών που πρέπει να γίνουν, και εργάτες-νήματα που εκτελούν μια λειτουργία πάνω στα στοιχεία της worklist. Αυτό που προσπαθεί να πετύχει το σύστημα Galois με αυτό τον τρόπο, είναι να πραγματοποιήσει παραλληλοποίηση ενός αλγορίθμου που είναι υλοποιημένος σε μια σειριακή λογική, όπως γίνεται στην γλώσσα SQL, όπου ο προγραμματιστής βλέπει ελάχιστες από τις λειτουργίες που γίνονται στο παρασκήνιο (sorts, joins, merges κλπ). Στόχος λοιπόν του Galois είναι να απαλλάξει τον προγραμματιστή από το βάρος της παραλληλοποίησης ενός αλγορίθμου.

Ο αλγόριθμος του Delaunay refinement για παράδειγμα, έχει ως κοινά δεδομένα μεταξύ των νημάτων τον γράφο και την worklist με τα ‘’κακά’’ τρίγωνα. Ο χρήστης γράφει τον κώδικα κατασκευής του γράφου, χρησιμοποιώντας τις κλάσεις του Galois, και του refinement ενός ‘’κακού’’ τριγώνου. Στην υλοποίηση αυτή δεν θα γίνει καμία αναφορά σε νήματα, κλειδώματα και παράλληλες εκτελέσεις, παρά μόνο η σχεδίαση του αλγορίθμου σε μια σειριακή worklist-based λογική. Στην συνέχεια ο προγραμματιστής θα υλοποιήσει μια for loop, όπου αλγόριθμος του refinement θα εφαρμοστεί σε όλη την worklist, με την διαφορά ότι αυτή η for loop επανάληψη δεν θα είναι δομή της γλώσσας, δηλαδή σειριακή, αλλά θα είναι μια δομή του Galois. Το σύστημα στην συνέχεια θα φροντίσει σε κάθε στοιχείο της worklist να εφαρμοστεί ο αλγόριθμος του refinement, αναθέτοντας τις εργασίες αυτές σε διαφορετικά νήματα. Με αυτό τον τρόπο το Galois στοχεύει στον διαχωρισμό των εργασιών υλοποίησης αλγορίθμου και υλοποίησης του παραλληλισμού σε δύο διαφορετικούς προγραμματιστές. Σε αυτό το μοντέλο, ο προγραμματιστής υλοποιεί τον αλγόριθμο με μια σειριακή λογική, χρησιμοποιώντας τις δομές του Galois, και άλλοι προγραμματιστές είναι υπεύθυνοι για τον παραλληλισμό των εργασιών.

Τα κομμάτια λοιπόν που απαρτίζουν το σύστημα Galois είναι οι i) Υψηλού επιπέδου αφηρημένες δομές και κλάσεις, με τις οποίες ο χρήστης κατασκευάζει τις δομές δεδομένων και τις συναρτήσεις του, ii) Βιβλιοθήκες που υλοποιούν τις δομές αυτές, τις οποίες έχουν υλοποιήσει οι προγραμματιστές του Galois και iii) το Runtime σύστημα που είναι υπεύθυνο για την ορθή εκτέλεση των προγραμμάτων του χρήστη, το οποίο είναι επίσης κομμάτι του Galois. Το παρών κεφάλαιο γράφτηκε σύμφωνα με τις δημοσιευμένες εργασίες και παρουσιάσεις πάνω στο σύστημα Galois.

4.1 Μοντέλο προγραμματισμού

Το κύριο στοιχείο στον τρόπο προγραμματισμού είναι οι *do_all* επαναλήψεις. Αυτές οι επαναλήψεις λαμβάνουν μια λίστα με στοιχεία και μια συνάρτηση, και εφαρμόζουν την συνάρτηση αυτή σε όλα τα στοιχεία της λίστας. Σε αντίθεση με τις *do_across* επαναλήψεις, οι *do_all* αποτελούνται από εργασίες που δεν έχουν καμία εξάρτηση στην σειρά εκτέλεσης τους, οπότε και μπορούν να εκτελεστούν με απολύτως παράλληλο τρόπο. Στην ίδια λογική, το Galois χρησιμοποιεί δύο iterators συνόλων, τους *unordered set iterators*, και *ordered set iterators*, οι οποίοι αναλαμβάνουν τον ρόλο των *do_all* και *do_across* δομών.

Και οι δύο αυτές δομές χρησιμοποιούνται πάνω σε thread safe worklists, που σημαίνει ότι κατά την διάρκεια της εκτέλεσης ενός iteration, νέες εργασίες μπορούν να εισέρχονται και να διαγράφονται από αυτήν. Τα αποτελέσματα των λειτουργιών αυτών γίνονται άμεσα διαθέσιμα στα υπόλοιπα νήματα. Σε αυτήν την λογική, η υλοποίηση του Delaunay mesh refinement αλγορίθμου εκπίπτει στον παρακάτω κώδικα:

```
Mesh m = InitialiseMesh();
```

```
Set worklist;
worklist.add(mesh.badTriangles());
for_each e in worklist do {
      if (e no longer in mesh) continue;
      Cavity c = new Cavity(e);
      c.Refine();
      worklist.add(c.badTriangles());
}
```

Στον παραπάνω κώδικα, για κάθε ''κακό'' τρίγωνο e, αρχικοποιείται η κοιλότητα στον γράφο που χρειάζεται επεξεργασία, εφαρμόζεται ο αλγόριθμος, και στο τέλος εισέρχονται τυχόν νέα ''κακά'' τρίγωνα στην worklist. Το γεγονός ότι η *for_each* ενθυλακώνει παραλληλοποίηση είναι απόλυτα αδιαφανές στον προγραμματιστή, καθώς ο παραπάνω κώδικας αποτελεί μια σωστή σειριακή υλοποίηση του αλγορίθμου. Η μόνη περίπτωση που χρήστης πρέπει να κάνει παραπάνω δουλειά είναι όταν τα αντικείμενα της worklist δεν μπορούν να επεξεργαστούν με οποιαδήποτε σειρά.

4.1.1 Μοντέλο μνήμης

Τα αντικείμενα που χρησιμοποιεί ο χρήστης στον κώδικα του, και γενικότερα όλα τα αντικείμενα του Galois, βρίσκονται στην κοινή μνήμη των νημάτων. Το σύστημα φροντίζει όταν ζητούνται νέοι πόροι, αυτοί να ζούνε στην κοινή μνήμη των νημάτων. Στο παραπάνω παράδειγμα τα αντικείμενα αυτά είναι το πλέγμα m, και η worklist. Οι εργάτες-νήματα εφαρμόζουν αλλαγές στα κοινά δεδομένα χρησιμοποιώντας μόνο τις συναρτήσεις των αντικειμένων σε έναν καθαρά object oriented προγραμματισμό και η συνέχεια της μνήμης μεταξύ διαφορετικών επεξεργαστών αφήνεται στο επίπεδο της cache. Στο παράδειγμα μας δηλαδή, η Cavity, που αποτελεί κλάση δημιουργημένη απόν τον χρήστη, χρησιμοποιεί τον γράφο μόνο μέσω συναρτήσεων που του δίνει η κλάση Mesh μέσω του αντικειμένου e. Αν και ο περιορισμός του προγραμματιστή στην χρήση της κοινής μνήμης μόνο μέσω των συναρτήσεων που προσφέρει το Galois φαίνεται αρκετά σημαντικός, στην πραγματικότητα αποτελεί λογική τεχνική. Η εισαγωγή ή η διαγραφή κόμβων ή ακμών από έναν γράφο για παράδειγμα, θα ήταν έτσι και αλλιώς μια κλήση μιας συνάρτησης πάνω σε ένα αντικείμενο. Αυτές τις απαραίτητες συναρτήσεις τις προσφέρει το Galois για τις δομές που χρησιμοποιεί. Με τον τρόπο αυτό το Galois μπορεί να υλοποιήσει με ευκολότερο τρόπο τον συγχρονισμό των νημάτων που γίνεται σε υψηλό επίπεδο, και θα συζητηθεί εκτενώς στο επόμενο κεφάλαιο.

4.1.2 Μοντέλο εκτέλεσης

Κατά την εκκίνηση του κώδικα του χρήστη, ένα κυρίαρχο νήμα ξεκινάει την εκτέλεση του σειριακού κώδικα και όλου του κώδικα εκτός των *do_all* επαναλήψεων. Όταν ξεκινάει η εκτέλεση μιας τέτοια επανάληψης, το κυρίαρχο νήμα δημιουργεί εργάτες-νήματα οι οποίοι αναλαμβάνουν να εκτελέσουν τις εργασίες μαζί με το κυρίαρχο νήμα. Η ανάθεση των εργασιών αφήνεται στον Scheduler, που είναι κομμάτι του runtime συστήματος, για σωστή κατανομή των εργασιών. Όλα τα νήματα που τρέχουν εργασίες μιας *do_all* επανάληψης, όταν τελειώσουν την δουλειά τους, συγχρονίζονται στο τέλος της επανάληψης μέσω ενός barrier. Δεδομένου αυτού του μοντέλου το κύριο πρόβλημα είναι η τελική εκτέλεση να αποτελεί μια σωστή σειριακή εκτέλεση. Το σύστημα πρέπει να φροντίσει ότι οι αναγνώσεις και οι εγγραφές είναι συγχρονισμένες, και αυτό απαιτεί και πληροφορία από τον χρήστη για την παραγωγή ενός σωστού σειριοποιήσιμου προγράμματος.

4.1.3 Iterators

Οι Set iterators που χρησιμοποιούνται από το Galois και γενικά η worklist εργασιών, αποτελούν μια κλασσική τεχνική προγραμματισμού irregular αλγορίθμων που έχει χρησιμοποιηθεί και στο παρελθόν σε εφαρμογές όπως η δημιουργία πλεγμάτων [37] και σε SAT λύτες [38]. Συνεπώς, η λογική του Galois είναι ίδια με την τακτική παραλληλισμού που ακολουθείται για την επίλυση τέτοιων προβλημάτων, οπότε και ο προγραμματισμός στο Galois δεν απαιτεί την δημιουργία του αλγορίθμου από την αρχή, αλλά απλά την μεταφορά του στις δομές δεδομένων του Galois.

Μια από τις πρώτες γλώσσες που χρησιμοποίησε Set Iterators είναι η SETL, μια γλώσσα φτιαγμένη για επιστημονικές πράξεις γύρω από την θεωρία συνόλων [39]. Στην γλώσσα αυτή, οι iterators κάνουν επαναλήψεις γύρω από αριθμούς με κάποιο βήμα [start:step:end], και ο κώδικας έχει την δυνατότητα να κάνει αλλαγές στο Set της επανάληψης, ωστόσο οι αλλαγές αυτές θα γίνουν ορατές μόλις τελειώσει η εκτέλεση του iteration, σε αντίθεση με τους Galois iterators.

Το Galois στην τωρινή του υλοποίηση περιλαμβάνει *do_all* επαναλήψεις στην μορφή των *ordered* και *unordered set iterators*. Μια πιο ολοκληρωμένη υλοποίηση αυτών, θα περιελάβανε iterators πάνω σε multi-set και maps αντικειμένων για παράδειγμα, καθώς στην τωρινή του υλοποίηση, οι set iterators μπορούν να χρησιμοποιηθούν μόνο πάνω σε worklists, οι οποίες δεν είναι παρά ένα wrapper μιας συνδεδεμένης λίστας. Η υποστήριξη αυτών των δομών, αλλά και επαναλήψεων πάνω σε ακεραίους στην [start:step:high] αναπαράσταση, όπως υπάρχει σε γλώσσες όπως η Matlab και η Fortran, είναι ένα χρήσιμο χαρακτηριστικό που λείπει από το Galois. Αν και η αναπαράσταση αυτή συχνά μεταφράζεται σε *do_all* loops για παράλληλη εκτέλεση, υπάρχει διαφορά μεταξύ αυτών των επαναλήψεων και των *set iterators* που χρησιμοποιεί τώρα το Galois. Για παράδειγμα, στο πολλαπλασιασμό πινάκων, στην i-j-k επανάληψη, οι i,j επαναλήψεις δεν έχουν καμία απολύτως εξάρτηση και μπορούν να είναι *do_all* loops ή *unordered set iterators*. Ωστόσο, η k επανάληψη δεν μπορεί να είναι *do_all*, καθώς απαιτείται σειριακή εκτέλεση και συνεπώς μπορεί μόνο να εκτελεστεί κάτω από έναν *ordered set iterator*.

Στο παραπάνω παράδειγμα ωστόσο, η υλοποίηση στο Galois θα απαιτούσε iterators εμφωλεύμενους ο ένας μέσα στον άλλον. Αν και η υποστήριξη τέτοιων δομών δεν αποτελεί πρόβλημα για την λογική του Galois, στην υλοποίηση του δεν το υποστηρίζει. Όταν ένα νήμα (που δεν είναι το κυρίαρχο νήμα) συναντά έναν iterator (ordered ή unordered) τότε τον εκτελεί σειριακά. Η εκτέλεση αυτή αποτελεί συνεπώς κομμάτι της παράλληλης εκτέλεσης του αρχικού iteration δημιουργώντας έτσι πιο χρονοβόρες παράλληλες εργασίες. Η κατάλληλη αντιμετώπιση εμφωλευμένων iterators δεν είναι πρόβλημα ορθότητας της εκτέλεσης, αλλά απόδοσης. Η υποστήριξη τέτοιων σχημάτων εγείρει διάφορα ερωτήματα όπως: τα νήματα-εργάτες που εκτελούν εργασίες του εξωτερικού iterator μπορούν να αναλάβουν εργασίες του εσωτερικού; Ποιοι από τους δύο iterators έχουν μεγαλύτερη προτεραιότητα στην εκτέλεση; Το Galois αποφασίζει να μην απαντήσει σε αυτά τα ερωτήματα στην τωρινή του υλοποίηση και εκτελεί τους εσωτερικούς iterators πάντα σειριακά.

4.2 Ιδιότητες ACID στο Galois

Όπως ειπώθηκε και στο προηγούμενο κεφάλαιο, η υλοποίηση της παράλληλης εκτέλεσης αφήνεται στο σύστημα και στις βιβλιοθήκες υλοποίησης των δομών του χρήστη. Το κύριο στοιχείο της παραλληλοποίησης στο Galois, είναι η εκτέλεση του κάθε iteration σαν δοσοληψία επί των κοινών δομών δεδομένων. Δηλαδή μια επανάληψη θα τρέξει, και στο τέλος θα κάνει commit τις αλλαγές της ή θα κάνει abort. Οι λειτουργίες αυτές είναι ανάλογες με αυτές στην εκτέλεση δοσοληψιών σε βάσεις δεδομένων, όπου και είναι απαραίτητο να

ικανοποιούνται οι ACID ιδιότητες. Η ερμηνεία των ACID ιδιοτήτων στο Galois είναι ίδια με αυτήν στις βάσεις δεδομένων.

Για να εκτελεστεί ένα iteration σαν δοσοληψία πρέπει το Galois σύστημα να διατηρεί και αυτό τις αντίστοιχες ιδιότητες. Ατομικότητα (Atomicity): Ένα iteration θα εκτελεστεί πλήρως ή καθόλου, Συνέπεια (Consistency): Οι δομές δεδομένων θα παραμείνουν σε συνεπή κατάσταση πριν και μετά την εκτέλεση, Απομόνωση (Isolation): Το κάθε iteration θα παράγει τα ίδια αποτελέσματα σαν να έτρεχε μόνο του, και Διάρκεια (Durability): Τα αποτελέσματα των iterations θα παραμείνουν διαθέσιμα σε περίπτωση σφάλματος.

Η ιδιότητα της διάρκειας δεν λαμβάνεται υπόψη στο Galois καθώς τα αντικείμενα είναι δομές στην κοινή μνήμη και δεν χρειάζεται να γράφονται σε δίσκο. Οι άλλες τρεις ιδιότητες ωστόσο είναι εξαιρετικά σημαντικές και γι' αυτό θα αναλυθούν στα επόμενα κεφάλαια.

4.2.1 Συνέπεια

Όλες οι δομές δεδομένων που χρησιμοποιούνται από τον χρήστη πρέπει να παραμένουν σε συνεπή κατάσταση σε οποιαδήποτε περίπτωση που ο χρήστης θα χρησιμοποιήσει κάποια μέθοδο αυτών. Για να ικανοποιηθεί αυτή η ιδιότητα πρέπει η κάθε μέθοδος πάνω σε ένα αντικείμενο να εκτελείται ατομικά. Με αυτό τον τρόπο η δομή δεδομένων είναι σειριοποιήσιμη και παραμένει σε συνεπή κατάσταση. Για να καταστεί μια δομή δεδομένων σειριοποιήσιμη μπορούν να χρησιμοποιηθούν διάφορες μέθοδοι. Ο πιο εύκολος τρόπος είναι χρησιμοποιηθεί ένα lock για την δομή ή αρκετά locks σε κάποια fine grain λογική. Εναλλακτικά μπορεί να χρησιμοποιηθεί transactional memory ή ατομικά block κώδικα αν το υποστηρίζει η γλώσσα.

Χρησιμοποιώντας σειριοποιήσιμα αντικείμενα (δομές δεδομένων) μειώνεται αρκετά η πολυπλοκότητα του αν δύο νήματα μπορούν να χρησιμοποιήσουν την δομή ταυτόχρονα, και να παράγουν σειριοποιήσιμο πρόγραμμα. Συνήθως για να ικανοποιηθεί αυτό πρέπει να λογαριαστούν όλες οι πιθανές σειρές εκτέλεσης των εντολών του προγράμματος. Ωστόσο, όταν τα αντικείμενα είναι σειριοποιήσιμα, τότε χρειάζεται να λογαριαστούν μόνο οι πιθανές εκτελέσεις των μεθόδων πάνω στα αντικείμενα αυτά. Στην περίπτωση του Galois, χρησιμοποιούνται κλειδώματα για να καταστούν οι μέθοδοι των αντικειμένων ατομικές, όχι όμως για τους γράφους. Τα κλειδώματα χρησιμοποιούνται μόνο στις δομές των worklist, όπου ο κώδικας των μεθόδων είναι μικρός, καθώς πρόκειται για *push*, *pop*, *size* μεθόδους σε *FIFO*, *LIFO* λίστες.

Στην περίπτωση των γράφων, δεν χρησιμοποιούνται κλειδώματα στις μεθόδους, αλλά το Runtime σύστημα ελέγχει για conflicts κρατώντας τον ''ιδιοκτήτη'' κάθε κόμβου. Κάθε κόμβος είναι ένα Lockable αντικείμενο με έναν ιδιοκτήτη-lock, όπου το ''lock'' υλοποιεί relaxed memory model πάνω σε μια ατομική μεταβλητή. Στην μεταβλητή αυτή γράφεται η διεύθυνση ενός αντικειμένου ThreadContext, το οποίο είναι το αντικείμενο που κρατάει τις πληροφορίες ενός iteration κατά την εκτέλεση του, το οποίο αλλάζει από νήμα σε νήμα. Όταν εκτελείται δηλαδή ένα iteration που πειράζει κάποιούς κόμβους, τότε πάνω στις αντίστοιχες μεταβλητές-ιδιοκτήτες των κόμβων αυτών γράφεται ''το νήμα'' που εκτελεί το iteration. Αν ένας κόμβος έχει ήδη ιδιοκτήτη κατά την εκτέλεση, τότε στέλνεται σήμα FAIL στο iteration, και αυτό κάνει abort. Αν αυτό δεν συμβεί τότε το iteration πάει για commit. Η μέθοδος σειριοποίησης των γράφων θα συζητηθεί περισσότερο στο Runtime κεφάλαιο.

4.2.2 Απομόνωση

Δεδομένης της ύπαρξης σειριοποιήσιμων αντικειμένων, το πρόβλημα που παραμένει στο Galois είναι αν μπορούν όλες οι σειρές εκτέλεσης συναρτήσεων πάνω σε αντικείμενα να διατηρήσουν ένα τελικά σειριοποιήσιμο πρόγραμμα. Ο προγραμματιστής δεν γνωρίζει την παράλληλη εκτέλεση του κώδικα, και συνεπώς δεν μπορεί να βοηθήσει στη σημασία των κλήσεων του.

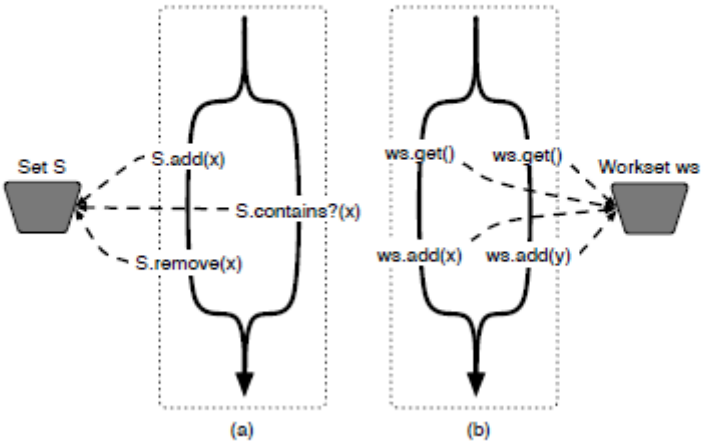

Figure 4: Κλήσεις συναρτήσεων από διαφορετικά iterations

Όπως φαίνεται στο παραπάνω παράδειγμα (a), αν το σύνολο S δεν περιέχει το x πριν την εκκίνηση των δύο επαναλήψεων, τότε υπάρχει μια πιθανή εκτέλεση που προκαλεί διαφορετικά αποτελέσματα. Η `contains(x)` θα έπρεπε να επιστρέψει `false`, αλλά αν εκτελεστεί όπως φαίνεται στην εικόνα τότε θα επιστρέψει `true`. Το κύριο πρόβλημα εδώ, είναι ότι η δεύτερη επανάληψη είναι σε θέση να δει αλλαγές από μια ενδιάμεση κατάσταση που έχει δημιουργήσει η πρώτη επανάληψη, η οποία δεν ξέρουμε ούτε αν θα κάνει commit. Όταν μια επανάληψη μπορεί και βλέπει ενδιάμεσα αποτελέσματα από μια άλλη επανάληψη τότε η σημασία της απομόνωσης έχει παραβιαστεί. Αυτό είναι και το πρόβλημα της απομόνωσης το Galois σύστημα. Ο προγραμματιστής μπορεί να μην τον ενδιαφέρει η σειρά της εκτέλεσης των κλήσεων του, ωστόσο το Galois πρέπει να φροντίσει ότι οι παράλληλες εκτελέσεις κλήσεων πάνω στα σειριοποιήσιμα αντικείμενα που προσφέρει, θα φέρουν τα ίδια αποτελέσματα σαν να εκτελούνταν η μια πίσω από την άλλη.

Συνήθως, η συνέπεια των αντικειμένων και η απομόνωση των επαναλήψεων γίνονται με τον ίδιο μηχανισμό. Θα μπορούσε μια επανάληψη να πάρει ένα lock, και να το αφήσει όταν έχει τελειώσει την δουλειά της. Αυτό όμως είναι αρκετά περιοριστικό από άποψη απόδοσης. Μια λύση είναι όλος ο κώδικας της επανάληψης να τρέχει σε transactional mode. Όπως είδαμε και στο προηγούμενο κεφάλαιο, αυτή λύση θα έλυνε όλα τα προβλήματα, και μπορεί να προσφέρει και καλή απόδοση.

Ωστόσο, αυτοί οι δύο τρόποι έχουν ένα κοινό μειονέκτημα. Ο προγραμματιστής, όπως ειπώθηκε, δεν βλέπει και δεν γράφει παράλληλο κώδικα. Αυτό σημαίνει ότι προσφέρει μόνο των κώδικα των iteration (που έχει κλήσεις στα αντικείμενα του Galois) στο Galois σύστημα. Αυτό σημαίνει ότι μια transactional εκτέλεση θα περιείχε όλον τον κώδικα μιας επανάληψης σε μια δοσοληψία. Άσχετα με τα προβλήματα απόδοσης που μπορεί να έχει αυτή η λύση, ας θυμηθούμε των κώδικα στο κεφάλαιο 4.1 του Delaunay Mesh Refinement. Σε κάθε επανάληψη διαβάζεται, και πιθανώς γράφεται με νέα ‘’κακά’’ τρίγωνα, η κοινή δομή worklist. Μπορεί τα τρίγωνα αυτά να μην έχουν καμία σχέση μεταξύ τους, ωστόσο το γεγονός αυτό μεταφράζεται κατευθείαν σε read, write conflict στο transactional memory καθώς οι pop, push λειτουργίες στην κοινή δομή από δύο διαφορετικά transactions θα διαβάζουν και θα γράφουν τις ίδιες μεταβλητές `head`, `tail`.

Το κύριο πρόβλημα είναι ότι οι κλήσεις πάνω στις κοινές δομές έχουν μια σημασιολογία η οποία δεν είναι διαθέσιμη σε μια υλοποίηση που δουλεύει κατευθείαν πάνω σε διευθύνσεις μνήμης. Το ζήτημα συνεπώς στο Galois είναι να μπορεί να καταλάβει την

σημασιολογία αυτή, και να μπορέσει να αξιοποιήσει το παραλληλισμό της με έναν αποδοτικό τρόπο.

4.2.2.1 Semantic commutativity

Για να λύσει αυτό το ζήτημα το Galois χρησιμοποιεί semantic commutativity μεταξύ των κλήσεων πάνω στα αντικείμενα. Δύο μέθοδοι μπορούν να ανταλλαχθούν μεταξύ τους, αν μπορούν να εκτελεστούν με οποιαδήποτε σειρά χωρίς να αλλάζουν τη σημασιολογική κατάσταση του αντικειμένου τους, και τα αποτελέσματα τους.

Έστω ότι έχουμε δύο επαναλήψεις, όπου η κάθε μια έχει έναν διαφορετικό αριθμό από κλήσεις αντικείμενων. Αν όλες οι κλήσεις της μιας επανάληψης μπορούν αντιμετατεθούν με αυτές της δεύτερης επανάληψης, τότε μπορούν να εκτελεστούν παράλληλα σε πλήρη απομόνωση. Δηλαδή, η αντιμεταθετικότητα των μεθόδων μεταξύ των δύο επαναλήψεων σημαίνει ότι οι κλήσεις σε αυτές τις μεθόδους μπορούν να γίνουν με οποιαδήποτε εμφωλευμένη σειρά, και να ικανοποιείται η ιδιότητα της σειριοποιησιμότητας. Το αποτέλεσμα θα είναι ίδιο με οποιαδήποτε σειρά εκτέλεσης. Αυτή η ιδιότητα μεταφέρεται και για αριθμό επαναλήψεων μεγαλύτερο των δύο.

Για παράδειγμα, στην Figure 4, η κλήση `contains(x)` δεν μπορεί να αντιμετατεθεί με τις κλήσεις `add(x)` και `remove(x)`, καθώς η διαφορετική σειρά εκτέλεσης φέρνει και διαφορετικά αποτελέσματα. Αντιθέτως, στο παράδειγμα (b), οι `get()` κλήσεις μπορούν να αντιμετατεθούν, και μια `get()` κλήση μπορεί να αντιμετατεθεί με μια `add(x)` αρκεί η τιμή επιστροφής της `get()` να μην είναι το x.

Είναι σημαντικό να σημειωθεί ότι η αντιμεταθετικότητα εδώ έχει σημασιολογική έννοια. Η διαφορετική σειρά εκτέλεσης των `add(x)`, `add(y)` έχει διαφορά στο Workset ws, αν αυτό είναι υλοποιημένο σαν συνδεδεμένη λίστα για παράδειγμα. Η πραγματική δομή εξαρτάται από την σειρά εκτέλεσης, καθώς κάποιο στοιχείο θα εισαχθεί πρώτο και κάποιο δεύτερο. Αυτό που έχει σημασία ωστόσο είναι ότι μετά το τέλος της εκτέλεσης το Workset ws θα έχει και το x, αλλά και το y. Μας ενδιαφέρει δηλαδή η κατάσταση του κοινού αντικειμένου σαν αφηρημένη δομή. Μια πιο αυστηρή αντιμεταθετικότητα θα ήταν η concrete commutativity, όπου έχει σημασία και ο τρόπος υλοποίησης της κοινής δομής. Αξίζει επίσης να σημειωθεί ότι το αν δύο μέθοδοι μπορούν να αντιμετατεθούν, εξαρτάται και από τα ορίσματα τους όπως φαίνεται παραπάνω. Δηλαδή μια `add` και μια `remove` μπορούν να αντιμετατεθούν μόνο αν τα ορίσματα τους είναι διαφορετικά.

Γενικά το πρόβλημα της σειριοποίησης δύο κομματιών κώδικα πρέπει να εξετάσει ότι όλες οι πιθανές σειρές εκτέλεσης μεταξύ των εντολών θα παράγουν το ίδιο αποτέλεσμα έτσι ώστε να μπορούν να εκτελεστούν παράλληλα. Ακόμη και με τις σειριοποιημένες μεθόδους του Galois, όπου το πρόβλημα ανάγεται σε σειρές εκτέλεσης μεθόδων, ο χώρος κατάστασης του προβλήματος αυτού είναι εκθετικός.

Ωστόσο, με semantic commutativity αρκεί κανείς να ελέγξει μόνο τα πιθανά ζευγάρια μεθόδων της συγκεκριμένης κλάσης. Κάθε μέθοδος πρέπει να ελεγχθεί για αντιμεταθετικότητα με κάθε άλλη μέθοδο του ίδιου αντικειμένου που έχει χρησιμοποιηθεί από άλλα uncommitted iterations την στιγμή της εκτέλεσης. Αυτό μειώνει τον χώρο κατάστασης από εκθετικό σε τετραγωνικό. Η τεχνική της σημασιολογικής αντιμεταθετικότητας ερευνήθηκε για πρώτη φορά το 1966 [40].

4.2.3 Ατομικότητα

Όπως έχει ειπωθεί και προηγουμένως, το Galois εκτελεί κάθε iteration ως δοσοληψία. Αυτό σημαίνει ότι αν δεν ανιχνευθεί κάποια σύγκρουση, τότε το iteration θα κάνει commit. Αν όμως ανιχνευθεί σύγκρουση τότε το σύστημα πρέπει να πάρει πίσω όλες τις αλλαγές που έχουν γίνει πάνω στα κοινά δεδομένα, και να εκτελέσει ξανά την δοσοληψία. Με τον τρόπο αυτό εξασφαλίζεται η ατομικότητα, δηλαδή ότι μια δοσοληψία θα εκτελεστεί πλήρως ή

καθόλου. Όταν ανιχνευθεί κάποια σύγκρουση λόγω του semantic commutativity, τότε το Galois ξεκινάει έναν μηχανισμό ανάκαμψης, έτσι ώστε η εκτέλεση να μπορεί να συνεχίσει. Επειδή λοιπόν το Galois υλοποιεί ένα αισιόδοξο μοντέλο παραλληλισμού, οι αλλαγές γίνονται κατευθείαν στα δεδομένα, και σε περίπτωση σύγκρουσης τότε αναιρεί τις αλλαγές.

Για να υλοποιηθεί αυτός ο μηχανισμός, κάθε μέθοδος κάποιας κλάσης κοινών δεδομένων (γράφων π.χ.) που επικαλείται ο προγραμματιστής, έχει μια αντίστοιχη *undo* μέθοδο. Κάθε *undo* μέθοδος, εφαρμόζει τις αντίστροφες αλλαγές πάνω στα κοινά δεδομένα. Για παράδειγμα, μια μέθοδος *add(x)* σε μια κλάση Set, έχει μια *remove(x)* μέθοδο για την undo διαδικασία, και η *remove(x)*, έχει την *add(x)*. Για το semantic commutativity, αυτό που έχει σημασία δεν είναι η πραγματική μορφή της δομής, όπως έχει αναφερθεί, αλλά η αναίρεση των αλλαγών να γίνει με σημασιολογική έννοια.

Όταν ένα iteration κάνει rollback λόγω σύγκρουσης, όλες οι undo μέθοδοι που επικαλείται, πρέπει να εκτελεστούν επιτυχώς. Αυτό σημαίνει ότι το σύστημα δεν πρέπει ποτέ να βρει σύγκρουση όταν εκτελεί undo μεθόδους. Αυτό εξασφαλίζεται με το να ελέγχεται η αντιμεταθετικότητα και με την undo μέθοδο και την κλήση μιας μεθόδου μιας κλάσης.

4.3 Κλάσεις αντικειμένων

Κάθε κλάση αντικειμένου της βιβλιοθήκης του Galois, πρέπει να έχει μεθόδους οι οποίες να μπορούν να εκτελεστούν μέσα σε ένα iteration, δηλαδή μέσα σε μια Galois δοσοληψία. Γι' αυτό το λόγο, το Galois χρησιμοποιεί wrappers κλάσεων. Καλώντας συναρτήσεις ενός απλά thread safe αντικειμένου μέσα σε μια δοσοληψία, αγνοούνται όλες οι διαδικασίες που αναφέρθηκαν στο προηγούμενο κεφάλαιο για να εξασφαλισθούν οι ACID ιδιότητες. Το Galois χρησιμοποιεί το design pattern delegate για αυτό το λόγο. Το thread safe αντικείμενο χρησιμοποιείται ως delegate, και όλες οι κλήσεις σε αυτό γίνονται μέσω του wrapper του.

```
class GaloisFoo {
	static final int METHOD_BAR = 1;
	public GaloisFoo(Foo delegate, ConflictDetection cd) {
		_cd = cd;
		_delegate = delegate;
	}
	public int bar(int a, int b) {
		_cd.prolog(METHOD_BAR, {a, b});
		int retval = bar(a);
		_cd.epilog({retval});
		GaloisRuntime.addUndo(/* ... */);
		return retval;
	}
	ConflictDetection _cd;
	Foo _delegate;
}
```

Όπως φαίνεται στον παραπάνω κώδικα, η GaloisFoo αποτελεί wrapper μιας κλάσης Foo, και η *bar* συνάρτηση της Foo καλείται μέσω της GaloisFoo, για να σιγουρευτεί η διαδικασία ανίχνευσης συγκρούσεων, και η προσθήκη της undo μεθόδου σε περίπτωση που χρειαστεί rollback. Το αντικείμενο ConflictDetection δέχεται πληροφορίες για την μέθοδο και τα ορίσματα της πριν την κλήση, για να κάνει τους ελέγχους commutativity. Επειδή οι πληροφορίες αυτές εξαρτώνται από την κλάση Foo και τις μεθόδους της, όλες οι κλάσεις αντικειμένων προσφέρουν μια διεπαφή με τρεις σημαντικές πληροφορίες για κάθε μέθοδο:

- *returns*: Ένα όνομα για την τιμή επιστροφής της μεθόδου
- *commutes*: Αναφέρει με ποιες άλλες μεθόδους μπορεί να γίνει αντιμετάθεση και κάτω από ποιες συνθήκες
- *undo*: Αναφέρει την αντίστροφη διαδικασία που της αντιστοιχεί

Έχοντας λοιπόν αυτές τις τρεις πληροφορίες για κάθε μέθοδο μιας thread safe κλάσης, κατασκευάζεται ένα Galois αντικείμενο το οποίο μπορεί να χρησιμοποιηθεί στον κώδικα των iteration. Οι συνθήκες που αναφέρονται στην *commutes*, δεν είναι παρά μια λογική έκφραση. Για παράδειγμα σε μια remove(x) μέθοδο, η διεπαφή θα μοιάζει με την παρακάτω:

```
void remove(Object x);
      [commutes]
            - add(y) {y != x}
            - remove(y) {y != x}
            - contains(y) {y != x}
            - findRandom() : y {y != x}
     [undo] add(x)
```

Αυτή η διεπαφή δεν είναι παρά ένα αρχείο που συνοδεύει την κλάση. Το γενικό πρόβλημα της μεταφοράς αυτών των εκφράσεων σε κώδικα αποτελεί ένα διαφορετικό πρόβλημα, και το Galois δεν ασχολείται αυστηρά με αυτό. Το πως το Galois υλοποιεί αυτήν την διαδικασία θα συζητηθεί στο κεφάλαιο για το Runtime σύστημα.

4.4 Galois Runtime

Το Runtime του Galois αποτελείται από τρία αντικείμενα: τον *scheduler* (δρομολογητή) που δημιουργεί τα iterations, τον *arbitrator* (διαιτητή) που αναλαμβάνει τον έλεγχο για τα aborts των iteration, και το *commit pool* που αναλαμβάνει το commit των iteration. Το Runtime αναλαμβάνει επίσης και όλους τους commutativity ελέγχους.

Ως σύνοψη, το σύστημα λειτουργεί ως εξής: Το commit pool κρατάει μια δομή για κάθε iteration που τρέχει. Αυτό το iteration record μοιάζει με το παρακάτω:

```
IterationRecord {
       Status status;
       Priority p;
       UndoLog ul;
       list<LocalConflictLog> local_log;
       Lock l;
}
```

Σε αυτήν την δομή, το Status ορίζει σε τι κατάσταση βρίσκεται μια δοσοληψία, το οποίο μπορεί να είναι μεταξύ των τιμών (RUNNING, RTC ή ABORTED), όπου RTC είναι ready to commit. Κάθε νήμα-εργάτης, που αναλαμβάνει να εκτελέσει ένα iteration, πηγαίνει στον Scheduler. Ο Scheduler, δημιουργεί ένα καινούριο iteration record, παίρνει το επόμενο στοιχείο από τον iterator, ορίζει την Priority μεταβλητή σύμφωνα με τον set iterator (για έναν unordered set iterator όλες οι προτεραιότητες είναι ίδιες), δημιουργεί χώρο στο commit pool, θέτει το Status σε RUNNING και επιστρέφει τον έλεγχο στο νήμα-εργάτη. Όταν ο κώδικας ενός iteration κάνει μια κλήση σε μια μέθοδο ενός Galois αντικειμένου, τότε το *local_log* ενημερώνεται, και η μέθοδος εισάγεται στο UndoLog αντικείμενο. Αν ανιχνευθεί μια commutativity σύγκρουση, τότε ο arbitrator αναλαμβάνει να προκαλέσει abort στο iteration με την χαμηλότερη προτεραιότητα, και να αφήσει το άλλο iteration να κάνει commit. Όλες οι κλήσεις στο UndoLog, του iteration που διακόπτεται, καλούνται σε LIFO σειρά για να αναιρέσουν τις μέχρι την στιγμή της εκτέλεσης αλλαγές. Όταν ένα νήμα τερματίσει την εκτέλεση ενός iteration, τότε το Status αλλάζει σε RTC, και το νήμα προχωράει να εκτελέσει την επόμενη επανάληψη. Όταν ένα RTC iteration έχει την μεγαλύτερη προτεραιότητα τότε μπορεί να κάνει commit.

4.4.1 Scheduler

Ο ρόλος του Scheduler είναι να βγάζει εργασίες από την λίστα, και να ετοιμάζει το περιβάλλον εκτέλεσης για τα νήματα-εργάτες. Στον τρόπο διαμοιρασμού μπορούν να εφαρμόζονται διάφορες πολιτικές, όπου η πιο εύκολη είναι η τυχαία στην οποία δεν λαμβάνεται κανένα κριτήριο υπόψη. Ωστόσο, μέσω της εξυπνότερης ανάθεσης εργασιών

είναι σίγουρο ότι μπορεί να αυξηθεί η απόδοση του προγράμματος, και γι' αυτό το λόγο το Galois χρησιμοποιεί διάφορες τεχνικές προς αυτήν την κατεύθυνση.

Ας πάρουμε για παράδειγμα τον αλγόριθμο του Delaunay Mesh refinement. Ένα νήμα που έχει αναλάβει ένα ''κακό'' τρίγωνο, έχει φέρει στην μνήμη του πολλά δεδομένα από τον γράφο, και θα ήταν πιο αποδοτικό το επόμενο τρίγωνο που θα αναλάβει να είναι κοντά στο προηγούμενο πάνω στον γράφο. Αν δηλαδή η επεξεργασία κάποιου τριγώνου έχει δημιουργήσει κάποιο νέο ''κακό'' τρίγωνο, τότε αυτό θα επεξεργαστεί αμέσως από το ίδιο νήμα που το εισήγαγε στην worklist. Αυτή η τακτική εκμεταλλεύεται την τοπικότητα των αναφορών του αλγορίθμου. Ένας αλγόριθμος δρομολόγησης που υλοποιεί μια απολύτως τυχαία επιλογή ''κακού'' τριγώνου αποδίδει έως και 30% χειρότερα στο πλαίσιο του Galois.

Από τα παραπάνω φαίνεται ότι το πρόβλημα της δρομολόγησης των set iterators στο Galois είναι πιο πολύπλοκο από αυτό των *do_all* επαναλήψεων σε regular αλγορίθμους. Γι' αυτό το λόγο, το Galois υλοποιεί Scheduler αντικείμενα, όπου το καθένα υλοποιεί διαφορετικούς αλγορίθμους δρομολόγησης για κάποιο συγκεκριμένο πρόβλημα. Τα σημεία που μπορεί να αξιοποιήσει ένας τέτοιος Scheduler είναι τα παρακάτω:

- Τοπικότητα: Για να αξιοποιηθεί η τοπικότητα των αναφορών στην μνήμη, είναι επιθυμητό τα iterations που διαβάζουν ένα συγκεκριμένο κομμάτι των κοινών δεδομένων, να εκτελούνται πάντα από τον ίδιο υπολογιστικό πυρήνα. Για παράδειγμα, όπως αναφέρθηκε και προηγουμένως, ''κακά'' τρίγωνα που είναι κοντά να αναθέτονται στο ίδιο νήμα.
- Συγκρούσεις: Επαναλήψεις που είναι πιθανό να προκαλέσουν σύγκρουση, είναι επιθυμητό να μην τρέξουν παράλληλα σε διαφορετικούς εργάτες. Αυτές οι επαναλήψεις θα πρέπει να αφήνονται να εκτελεστούν από το νήμα που έχει ''αναλάβει'' εκείνο το κομμάτι των κοινών δεδομένων.
- Ισοκατανομή φόρτου: Η ανάθεση της δουλειάς πρέπει να δημιουργεί ισοκατανομή του συνολικού φόρτου, ειδικά σε ένα πρόβλημα όπου η δουλειά δημιουργείται δυναμικά. Επίσης ένα σωστό load balancing μπορεί να έρθει σε αντίθεση με την τοπικότητα των αναφορών και την αποφυγή συγκρούσεων. Ένα τέλειο load balancing είναι να δίνεται κάθε φορά ένα τυχαίο τρίγωνο για παράδειγμα.

Αυτά τα τρία χαρακτηριστικά, καθιστούν την δρομολόγηση που γίνεται στις *do_all* επαναλήψεις να μην είναι αρκετή. Στο OpenMP για παράδειγμα, υπάρχουν οι static, dynamic και guided τεχνικές για δρομολόγηση. Στην δυναμική δρομολόγηση, οι εργασίες αναθέτονται σε δουλειές από πριν με κυκλικό τρόπο, ή σε blocks για καλύτερη τοπικότητα. Στους irregular αλγορίθμους όμως η δουλειά δημιουργείται δυναμικά. Στην δυναμική δρομολόγηση, σε κάθε πυρήνα δίνεται δουλειά, όταν αυτός έχει τελειώσει. Με αυτήν την πολιτική υπάρχει καλή κατανομή εργασίας, αλλά δεν γίνεται εκμετάλλευση των δύο πρώτων, πολύ σημαντικών, σημείων.

Το Galois υλοποιεί τρείς τεχνικές για να χωρίσει και να μοιράσει των χώρο των επαναλήψεων:

- Clustering: Δημιουργούνται ομάδες από iterations όπου η κάθε ομάδα θα εκτελεστεί σε έναν συγκεκριμένο πυρήνα.
- Labeling: Ορίζει ποιος πυρήνας θα αναλάβει ποιο cluster επαναλήψεων. Ένας πυρήνας μπορεί να αναλάβει παραπάνω από ένα cluster.
- Ordering: Για κάθε cluster που υπάρχει, δημιουργείται η σειρά με την οποία θα εκτελεστούν τα iterations.

Οι τεχνικές αυτές είναι προσαρμοσμένες για δυναμική δρομολόγηση. Δηλαδή, οι αρχικές εργασίες θα χωριστούν και θα δρομολογηθούν, και στην συνέχεια καθώς εισέρχονται νέα εργασίες στην λίστα, μπορεί να τρέξουν ξανά κάποιοι από τους παραπάνω αλγορίθμους, και να δημιουργήσουν πιθανώς και νέα clusters.

4.4.1.1 Clustering

Στο Galois υπάρχουν πέντε τεχνικές για το clustering των εργασιών:

- Random: Η ομαδοποίηση γίνεται με τυχαίο τρόπο. Ο αριθμός των clusters μπορεί να ορίζεται από τον χρήστη, ή να είναι ίσος με τον αριθμό των πυρήνων του συστήματος.
- Chunking: Η τεχνική αυτή μοιάζει με αυτήν του OpenMP, όπου ο προγραμματιστής ορίζει το μέγεθος των clusters και το σύστημα ομαδοποιεί συνεχόμενα iterations μαζί.
- Data-centric: Η τεχνική αυτή υλοποιείται όταν υπάρχει μια κοινή δομή δεδομένων την οποία θα διαβάσουν και θα γράψουν τα νήματα κατά την επανάληψη. Η τεχνική αυτή έχει σκοπό να εκμεταλλευτεί την τοπικότητα των αναφορών σε κάθε πυρήνα. Ο αριθμός των clusters εδώ μπορεί να βρεθεί και από το ίδιο το σύστημα ευριστικά.
- Unit: Κάθε iteration βρίσκεται σε ένα cluster μόνη της.
- Inherited: Η τεχνική αυτή υπάρχει για να δημιουργήσει την υποστήριξη της δυναμικής προσθήκης εργασιών. Μια εργασία που έχει προκύψει δυναμικά, θα ομαδοποιηθεί στο cluster στο οποίο είναι η εργασία που την δημιούργησε.

Μια εργασία που έχει σταματήσει την εκτέλεση της λόγω abort, αντιμετωπίζεται ως καινούρια δημιουργημένη εργασία, και θα ομαδοποιηθεί στο cluster στο οποίο ήταν εξ αρχής.

Η λογική του Clustering που υλοποιεί το Galois μοιάζει με την τεχνική *owner-computes* [41]. Στην *owner-computes*, τα δεδομένα χωρίζονται σε ομάδες τις οποίες αναλαμβάνει ένας συγκεκριμένος φυσικός επεξεργαστής. Ωστόσο, η inherited τεχνική αναθέτει δουλειά σε ομάδες σύμφωνα με το ποιος πυρήνας δημιούργησε την δουλειά, και όχι σύμφωνα με τη ‘’θέση΄΄ της δουλειάς στα κοινά δεδομένα. Μπορεί στο Delaunay refinement να καταλήγει τελικά στην ίδια ομάδα, ωστόσο υπάρχει η δυνατότητα να ομαδοποιηθεί σε διαφορετικό cluster σε κάποιον άλλον αλγόριθμο. Αυτό το σημείο είναι και αυτό που διαφοροποιεί το clustering σε σχέση με το *owner-computes*.

4.4.1.2 Labeling

Στο Galois υπάρχουν πέντε διαφορετικές τεχνικές για το labeling των clusters.

- Random: Οι δημιουργημένες ομάδες αναθέτονται τυχαία σε λογικούς πυρήνες
- Round-Robin: Οι δημιουργημένες ομάδες αναθέτονται σε φυσικούς πυρήνες με Round-Robin σειρά, όπως γίνεται και στο OpenMP σε επίπεδο επαναλήψεων.
- Data-centric: Αν το clustering έχει γίνει με Data-centric λογική τότε η ομάδα ανατίθεται στον επεξεργαστή που έχει τα δεδομένα.
- LIFO/FIFO: Όταν δημιουργείται δυναμικά ένα καινούριο cluster, τότε αν ένας πυρήνας χρειάζεται δουλειά, του ανατίθεται το νέο cluster.
- Data-centric-partitioning: Όταν ένας πυρήνας χρειάζεται δουλειά, τότε ένα υπάρχον cluster σπάει σε μικρότερα, χρησιμοποιώντας data-centric λογική, και τα νέα clusters ανατίθενται στους πυρήνες.

4.4.1.3 Ordering

Το Galois χωρίζει το Ordering σε δύο διαφορετικά κομμάτια. Δεδομένου ότι σε ένα νήμα έχουν ανατεθεί παραπάνω από ένα clusters, υπάρχουν δύο πολιτικές. Η Inter-cluster ordering για το πότε αλλάζει το cluster από το οποίο εκτελούνται iterations, και η intra-cluster ordering για την σειρά των iterations μέσα στο cluster.

- Random: Η επιλογή του cluster γίνεται τυχαία, και η εκτέλεση μεταξύ τους αλλάζει επίσης τυχαία.
- Lexicographic: Όταν μιλάμε για ordered-set iterator, η εκτέλεση γίνεται σύμφωνα με την σειρά στον χώρο των επαναλήψεων.
- Cluster-major: Κάθε cluster εκτελείται πλήρως πριν η εκτέλεση περάσει στο επόμενο
- Switch-on-abort: Η εκτέλεση πάει στο επόμενο cluster, όταν υπάρχει κάποιο abort.

Όταν ένα cluster επιλεχθεί, η intra-cluster πολιτική μπορεί να είναι μια από τις παρακάτω:

- Random: Τα iterations εκτελούνται τυχαία.
- Lexicographic: Η σειρά εκτέλεσης ορίζεται από τον ordered-set iterator
- LIFO/FIFO: Η δυναμική προσθήκη εργασιών στο cluster εξυπηρετείται σε LIFO/FIFO σειρά.

4.4.2 Arbitrator

Ο ρόλος του Arbitrator στο Galois είναι να επιλύει τις συγκρούσεις μεταξύ των iteration. Όταν η επανάληψη είναι πάνω σε έναν unordered set iterator, τότε από άποψη ορθότητας, η επιλογή του iteration που θα συνεχίσει την επανάληψη δεν έχει καμία σημασία. Ωστόσο, από άποψη απόδοσης, η επιλογή μπορεί να επηρεάζει σημαντικά την εκτέλεση καθώς οι συνεχείς προσπάθειες εκτέλεσης δοσοληψιών που έχουν εξαρτήσεις προκαλεί συνεχώς συγκρούσεις.

Στην υλοποίηση πολιτικών για τον ρόλο του Arbitrator, έχουν προταθεί αρκετές πολιτικές που εφαρμόζονται κυρίως στο transactional memory. Αυτές οι πολιτικές χρησιμοποιούν διάφορες μετρικές όπως την ‘’ηλικία’’ μιας δοσοληψίας, τον χώρο στην μνήμη που χρησιμοποιεί, ή και τυχαία επιλογή [42]. Στο Galois ωστόσο υλοποιείται μια μόνο πολιτική, το νήμα που εκτελεί ένα iteration και βρίσκει μια σύγκρουση σταματάει την εκτέλεση του.

Αυτή η πολιτική ωστόσο, έχει ένα πρόβλημα. Στην περίπτωση του ordered set iterator, αν ένα iteration (i1) που έχει μεγαλύτερη προτεραιότητα, βρει μια σύγκρουση με ένα iteration (i2) που έχει μικρότερη, τότε αυτή η πολιτική δεν μπορεί να εφαρμοστεί έτσι απλά. Αν το i2 συνεχίσει την εκτέλεση του, και το i1 σταματήσει, τότε το i2 θα πάει για commit. Όταν το i1 εκτελεστεί ξανά θα δει την σύγκρουση με το i2, που δεν έχει κάνει commit λόγω της μικρότερης προτεραιότητας, και θα ξανακάνει abort. Αυτή η αλληλουχία οδηγεί το σύστημα σε live-lock. Γι’ να αποφευχθεί αυτό, σε αυτήν μόνο την περίπτωση ο Arbitrator σταματάει την εκτέλεση του i2.

Ωστόσο, αξίζει να σημειωθεί εδώ ότι επειδή το νήμα που εκτελεί το i1 είναι αυτό που έχει εντοπίσει την σύγκρουση, δεν υπάρχει προφανής τρόπος να σταματήσει το νήμα που εκτελεί την i2, και να του επιβάλλει να κάνει abort. Γι’ αυτό το λόγο, το i1 είναι αυτό που θα εκκινήσει την διαδικασία του abort με το να θέσει το status της i2 σε ABORTED, και να εκτελέσει το undo log της. Στην συνέχεια η i1 θα συνεχίσει την εκτέλεση της, και η i2 θα σταματήσει μόνη της όταν προσπαθήσει να διαβάσει ή να γράψει στα κοινά δεδομένα, όπου και θα δει το αλλαγμένο status της.

Ένα βασικό κομμάτι της rollback διαδικασίας είναι η εκτέλεση του undo-log. Μια undo συνάρτηση έχει στο σώμα της προσβάσεις στα κοινά δεδομένα. Ωστόσο, όταν φτάσει η ώρα να εκτελεστούν οι undo κλήσεις ο γράφος και οι συνδέσεις του μπορεί να έχουν αλλάξει μορφή. Για να λύσει αυτό το πρόβλημα το Galois, πρέπει να φροντίσει όλα τα iteration που πρέπει να κάνουν abort, να το κάνουν σε LIFO σειρά. Ένα δεύτερο πρόβλημα είναι η αποδέσμευση μνήμης. Επειδή δεν υπάρχει τρόπος να γίνει undo μια τέτοια διαδικασία, όλες οι κλήσεις για αποδέσμευση μνήμης γίνονται πάντα στο τέλος μιας επανάληψης, και αφού αυτή έχει κάνει commit επιτυχώς.

4.4.3 Commit pool

Ο ρόλος του commit pool είναι να ολοκληρώνει την εκτέλεση ενός iteration. Όταν η επανάληψη είναι πάνω σε έναν unordered set iterator, τότε όλες οι δοσοληψίες έχουν την ίδια προτεραιότητα. Αυτό δεν ισχύει σε έναν ordered set iterator. Όταν εκτελείται ένα ordered set, πρέπει να ικανοποιείται μια συγκεκριμένη σειρά. Γι' αυτό το λόγο η σειρά των commit φροντίζεται από το commit pool.

Ο ρόλος του commit pool μοιάζει αρκετά με τον reorder buffer στα pipeline των επεξεργαστών. Οι εντολές εκτελούνται με όποιο τρόπο θέλουν, αλλά το commit τους γίνεται με μια συγκεκριμένη σειρά. Κατ' αντιστοιχία, όλες οι δοσοληψίες μπορούν να εκτελεστούν ταυτόχρονα, ωστόσο θα παραμείνουν στο commit pool μέχρι αυτό να περιμένει όλες τις δοσοληψίες με μεγαλύτερη προτεραιότητα. Η διαφορά με τον reorder buffer είναι ότι τα αποτελέσματα της εκτέλεσης μια δοσοληψίας είναι ορατά στις υπόλοιπες. Απλά η παραμονή της στο commit pool γίνεται για να προκληθεί abort άμα εκτελεστεί δοσοληψία με μεγαλύτερη προτεραιότητα που έχει κάποια εξάρτηση.

Για να το υλοποιήσει αυτό, το commit pool διατηρεί μια commit queue δομή, στην οποία υπάρχουν εγγραφές για όλα τα iteration που τρέχουν, ταξινομημένες κατά προτεραιότητα. Commit κάνει πάντα το iteration που είναι στην κορυφή, και μόνο αν τα iterations που έχουν μείνει για εκτέλεση έχουν όλα μικρότερη προτεραιότητα. Η τελευταία συνθήκη είναι έτσι ώστε οι επαναλήψεις που είναι να εκτελεστούν μετά από αυτή, σύμφωνα με την σειρά του ordering αλγορίθμου, να μην δημιουργήσουν κάποιο νέο iteration με μεγαλύτερη προτεραιότητα.

Η επεξεργασία της commit queue γίνεται lazily. Όταν κάποιο νήμα που εκτελεί ένα iteration δει ότι μπορεί να το κάνει commit, τότε κάνει commit και οποιοδήποτε άλλο iteration μπορεί μέσα στην commit queue. Το νήμα αυτό βγάζει επίσης από την commit queue και όποια άλλη ABORTED δοσοληψία που είναι στην κορυφή και στην συνέχεια συνεχίζει να εκτελέσει κάποιο καινούριο iteration.

4.4.4 Conflict logs

Το conflict log είναι η υλοποίηση του ConflictDetection αντικειμένου που φαίνεται στον κώδικα στην ενότητα 4.3. Η δουλειά του είναι να κάνει τους semantic commutativity ελέγχους. Το conflict log ενός αντικειμένου είναι μια δομή που περιλαμβάνει, για όλες τις τρέχουσες δοσοληψίες, τις κλήσεις μεθόδων πάνω σε αυτό το αντικείμενο. Οι εγγραφές αυτές περιλαμβάνουν το όνομα της μεθόδου, καθώς επίσης και τα ορίσματα της. Όταν γίνεται μια νέα κλήση κάποιας μεθόδου του αντικειμένου αυτού, ελέγχονται όλες οι εγγραφές της δομής, και αν οι δύο κλήσεις δεν μπορούν να αντιμετατεθούν για κάποια εγγραφή, ξεκινάει η διαδικασία του abort. Αν η νέα κλήση αντιμετατίθεται με όλες τις ήδη υπάρχουσες μέσα στην δομή, τότε η νέα αυτή κλήση εισέρχεται στην δομή. Όταν ένα iteration κάνει abort ή commit, τότε όλες οι εγγραφές μεθόδων διαγράφονται από την δομή.

Μια βελτιστοποίηση αυτής της τεχνικής είναι να κρατούνται διαφορετικά sets με εγγραφές, ένα για κάθε μέθοδο του αντικειμένου. Σε αυτό το set υπάρχουν όλες οι κλήσεις που έγιναν και είναι πιθανό να προκαλέσουν σύγκρουση. Όταν γίνεται μια νέα κλήση, τότε μόνο το συγκεκριμένο set χρειάζεται να ελεγχθεί. Μια ακόμη βελτιστοποίηση που εφαρμόζει το Galois είναι το `local_log` που φαίνεται στο IterationRecord στην ενότητα 4.4. Σε αυτήν την δομή το iteration κρατάει ''cache'' με όλες τις επιτυχημένες κλήσεις του πάνω σε ένα αντικείμενο. Όταν γίνεται μια νέα κλήση, τότε πρώτα ελέγχεται το `local_log`, και αν τα περιεχόμενα του δείχνουν ότι η κλήση μπορεί να γίνει τότε δεν ελέγχει το conflict log του αντικειμένου.

4.5 Σύνοψη

Το Galois σύστημα συνδυάζει μια πληθώρα τεχνικών και αλγορίθμων. Υλοποιεί worklist-based αλγορίθμους όπου η δουλειά που πρέπει να γίνει χωρίζεται σε εργασίες. Υλοποιεί αισιόδοξο παραλληλισμό δημιουργώντας ένα μοντέλο εκτέλεσης, σαν αυτό των βάσεων δεδομένων, όπου κάθε διαθέσιμη εργασία εκτελείται από κάποιο νήμα εργάτη. Φροντίζει ώστε οι εκτελέσεις των εργασιών να μην παραβιάζουν τις ACID ιδιότητες, χωρίς να επιβαρύνει τον χρήστη με την συγγραφή ‘’παράλληλου’’ κώδικα. Εφαρμόζει έξυπνους αλγορίθμους δρομολόγησης των εργασιών και διακοπής εργασιών στις οποίες ανιχνεύονται συγκρούσεις. Υλοποιεί ένα μοντέλο ανίχνευσης συγκρούσεων στους γράφους που δεν απαιτεί κλειδώματα. Προσφέρει επίσης έτοιμες υλοποιήσεις αντικειμένων, όπως worklist και γράφους, στον προγραμματιστή για να υλοποιήσει τους αλγορίθμους του.

Το κεντρικό στοιχείο που έχει το Galois σύστημα είναι η διάκριση των αρμοδιοτήτων των προγραμματιστών. Στόχος του Galois είναι ο χρήστης του να μην γράφει ούτε μια γραμμή παράλληλο κώδικα, και να τα αφήνει όλα στο σύστημα. Στο trade-off που υπάρχει δηλαδή μεταξύ προγραμματιστικού κόπου, και απόδοσης του μηχανισμού παραλληλοποίησης (coarse grain, file grain, lock free…), το Galois αποφασίζει να προσφέρει στους χρήστες ένα διαφορετικό, εύκολο προγραμματιστικό μοντέλο, αφήνοντας την υλοποίηση του συστήματος παράλληλης εκτέλεσης σε πολύπειρους προγραμματιστές. Αυτό προσφέρει πολύ καλή απόδοση στην εκτέλεση του κώδικα, ωστόσο για να μπορέσει να υλοποιηθεί στην πράξη, οι δομές που χρησιμοποιεί ο χρήστης πρέπει να είναι όλες Galois δομές. Πρέπει να ξέρει το Galois, ανά πάσα στιγμή, όλες τις συναρτήσεις που μπορεί να χρησιμοποιηθούν επί των κοινών δεδομένων από τα νήματα εργάτες. Αυτό σημαίνει για τον χρήστη ότι πρέπει να χτίσει τον κώδικα του, όχι στις βιβλιοθήκες γράφων που ξέρει για παράδειγμα, αλλά σε αυτές που προσφέρει το Galois.

Αυτό που είναι ξεκάθαρο, είναι η δυσκολία υλοποίησης ενός συστήματος σαν το Galois. Αυτή η δυσκολία ωστόσο, δεν μεταφέρεται στον προγραμματιστή. Στην υλοποίηση αλγορίθμων στο Galois, ο χρήστης δεν ασχολείται καθόλου με την παραλληλοποίηση. Ωστόσο, είναι ξεκάθαρο ότι οι εργασίες του αλγορίθμου πάνω στα κοινά δεδομένα, μοντελοποιούνται ως δοσοληψίες. Γι’ αυτό το λόγο το Galois προσφέρει υποστήριξη στους χρήστες που θέλουν να τρέξουν τα προγράμματα τους κάτω από transactional memory, με software transactional memory, αν το υποστηρίζει ο compiler. Στα πλαίσια αυτής της εργασίας, γίνεται χρήση hardware transactional memory για την υλοποίηση της Ατομικότητας, Συνέπειας και Απομόνωσης των Galois δοσοληψιών.

Στα επόμενα κεφάλαια λοιπόν, θα αναλυθούν οι αλγόριθμοι που χρησιμοποιήθηκαν και η υλοποίηση τους σε transactional memory, και θα συζητηθούν τα αποτελέσματα απόδοσης τους σε σχέση με αυτά του Galois.

5 Case Studies

Για την αξιολόγηση του συστήματος Galois και των μεθόδων συγχρονισμού του, σε σχέση με το Transactional Memory επιλέχθηκαν 4 αλγόριθμοι. Οι εκτελέσεις των προγραμμάτων έγιναν σε δύο διαφορετικά μηχανήματα:

- broady: Το σύστημα broady αποτελείται από Intel® Xeon® CPU E5-2699 v4 @ 2.20GHz επεξεργαστές. Το chip περιέχει 22 φυσικούς πυρήνες αρχιτεκτονικής Broadwell και 64Gb φυσικής μνήμης. Εξυπηρετεί 44 νήματα συνολικά μέσω της τεχνολογίας HyperThreading®. Το σύστημα αποτελείται από δύο επεξεργαστές φτάνοντας συνολικά τα 88 νήματα. Κάποιες εκτελέσεις έχουν γίνει μέχρι τα 44 νήματα. Σε αυτές τις εκτελέσεις είναι απενεργοποιημένο το HyperThreading.
- haci3: Το σύστημα haci3 αποτελείται από Intel® Xeon® CPU E5-2697 v3 @ 2.60GHZ επεξεργαστές. Το chip περιέχει 14 φυσικούς πυρήνες αρχιτεκτονικής Haswell και 128Gb φυσικής μνήμης. Εξυπηρετεί 28 νήματα συνολικά μέσω της τεχνολογίας HyperThreading®. Το σύστημα αποτελείται από δύο επεξεργαστές φτάνοντας συνολικά στα 56 νήματα.

Για την παρουσίαση των αποτελεσμάτων των εκτελέσεων χρησιμοποιούνται τέσσερεις διαφορετικοί τύποι διαγραμμάτων.

- acc: Διάγραμμα επιτάχυνσης όπου κάθε σημείο είναι η διαίρεση του χρόνου εκτέλεσης ενός νήματος, προς τον χρόνο εκτέλεσης πολλών νημάτων
- transactions committed: Διάγραμμα επιτυχώς ολοκληρωμένων transactions σε απόλυτο αριθμό
- committed percentage: Διάγραμμα ποσοστών για επιτυχώς ολοκληρωμένα transactions σε σχέση με τις συνολικές προσπάθειες
- aborts: Σύγκριση απόλυτου αριθμού aborts

5.1 Barnes-Hut

Ο αλγόριθμος Barnes Hut χρησιμοποιείται για την προσομοίωση της κίνησης ουράνιων σωμάτων σε ένα σύμπλεγμα ενός γαλαξία, και τον υπολογισμό των βαρυτικών δυνάμεων που ασκούνται μεταξύ τους, χρησιμοποιώντας την λογική του n-body simulation για ταχύτερη εκτέλεση του αλγορίθμου [43]. Χρησιμοποιώντας αυτόν τον αλγόριθμο για τον υπολογισμό, αποφεύγεται ο έλεγχος όλων των συνδυασμών των σωμάτων μεταξύ τους, και η πολυπλοκότητα εκπίπτει σε $nlogn$. Για να το πετύχει αυτό, ο αλγόριθμος χωρίζει τον χώρο των σωμάτων σε κυβικά κελιά, μέσω μιας δομής $octree$. Το $octree$ αποτελεί μια επέκταση του δυαδικού δέντρου στον τρισδιάστατο χώρο, έτσι ώστε αντικείμενα που βρίσκονται μακριά το ένα από το άλλο, να βλέπονται ως κυβικά κελιά, δηλαδή ομάδες αντικειμένων με κοινό κέντρο μάζας και ταχύτητα, και η συνδρομή τους στον υπολογισμό της δύναμης του μακρινού αντικειμένου να γίνεται ομαδικά. Αντιθέτως, για τα αντικείμενα που βρίσκονται κοντά, η συνδρομή τους στην συνολική δύναμη υπολογίζεται ξεχωριστά για το καθένα. Αυτό μειώνει τον χώρο των υπολογισμών από όλους τους δυνατούς συνδυασμούς $O(n^2)$, σε O(nlogn).

Η δομή octree είναι ιεραρχική. Δηλαδή, τα σώματα ομαδοποιούνται ιεραρχικά σε ομάδες σωμάτων όπου κάθε τέτοια ομάδα είναι ένας εσωτερικός κόμβος στο δέντρο. Κάθε τέτοιος κόμβος αναπαρίσταται με το κέντρο βαρύτητας και την συνδυασμένη μάζα των σωμάτων που βρίσκονται χαμηλότερα στην ιεραρχία του δέντρου. Ο πρώτος κόμβος του δέντρου αναπαριστά τον συνολικό χώρο που καταλαμβάνουν τα σώματα, και κάθε ένα από τα 8 παιδιά του αναπαριστά ένα όγδοο του συνολικού χώρου.

Για τον υπολογισμό της δύναμης που ασκείται σε ένα σώμα, ο αλγόριθμος ξεδιπλώνει το δέντρο και αρχίζει από τον πρώτο-κεντρικό κόμβο. Αν το κέντρο μάζας του κόμβου είναι αρκετά μακριά από το σώμα που μας ενδιαφέρει τότε συνυπολογίζεται δύναμη που ασκεί η

ομάδα αυτή. Αν το κέντρο μάζας είναι κοντά, τότε η διαδικασία αυτή συνεχίζεται στα παιδιά του κόμβου. Για τον υπολογισμό της απόστασης χρησιμοποιείται το πλάτος της ομάδας των σωμάτων προς την απόσταση του κέντρου μάζας της ομάδας αυτής σε σχέση με το εξεταζόμενο σώμα. Η ποσότητα αυτή συγκρίνεται με μια τιμή ϑ, η οποία προσδιορίζεται ανάλογα με την ακρίβεια που απαιτείται στον υπολογισμό. Για ϑ ίσο με 0, ο αλγόριθμος εκπίπτει σε $O(n^2)$.

Η διαδικασία του υπολογισμού είναι επαναληπτική, δηλαδή ο υπολογισμός των δυνάμεων γίνεται σε συνεχόμενα ίδια βήματα μέχρι οι αλλαγές στις δυνάμεις να είναι αρκετά μικρές. Η υλοποίηση που χρησιμοποιείται σε μορφή worklist είναι η παρακάτω:

```
List bodylist = ...
foreach timestep do {
	Octree octree;
	foreach Body b in bodylist {
		octree.Insert(b);
	}
	octree.SummarizeSubtrees();
	foreach Body b in bodylist {
		b.ComputeForce(octree);
	}
	foreach Body b in bodylist {
		b.Advance();
	}
}
```

Πρώτα αρχικοποιείται η λίστα με τα σώματα και τα ακριβή δεδομένα τους, και στην συνέχεια εφαρμόζεται μια επανάληψη στον χρόνο. Σε κάθε επανάληψη δημιουργείται ένα νέο octree εισάγοντας όλα τα σώματα στην δομή, και υπολογίζοντας ιεραρχικά τις πληροφορίες των κεντρικών κόμβων-ομάδων. Στην συνέχεια ο υπολογισμός χωρίζεται σε δύο στάδια. Στο πρώτο στάδιο υπολογίζεται η δύναμη που ασκείται σε κάθε σώμα, και στο δεύτερο στάδιο ενημερώνονται οι τιμές πάνω στα σώματα. Η πρώτη επανάληψη διαβάζει από τα κοινά δεδομένα, και η δεύτερη επανάληψη γράφει τις αλλαγές.

5.1.1 Galois

Οι δύο κύριες δομές που χρησιμοποιούνται από το Galois για τον παραλληλισμό αυτού του αλγορίθμου είναι unordered λίστες που διαβάζονται από έναν unordered set iterator, και το octree. Από τις 5 επαναλήψεις που υπάρχουν στον προηγούμενο κώδικα (η SummarizeSubtrees(…) που υπολογίζει τους ενδιάμεσους κόμβους είναι και αυτή μια επανάληψη), παραλληλοποιούνται όλες εκτός από την επανάληψη στον χρόνο. Το χτίσιμο του octree παραλληλοποιείται καθώς τα σώματα μπορούν να εισαχθούν στην δομή με οποιαδήποτε σειρά. Ο ιεραρχικός υπολογισμός των κεντρικών κόμβων, ο υπολογισμός των δυνάμεων και η ενημέρωση των κόμβων για την επόμενη χρονική στιγμή παραλληλοποιούνται επίσης χωρίς να υπάρχει κάποια εξάρτηση στην σειρά με την οποία θα επεξεργαστούν οι κόμβοι. Για τον λόγο αυτό, όλες οι επαναλήψεις υλοποιούνται με unordered set iterator στο Galois. Τρέχοντας τον Barnes Hut αλγόριθμο στο σύστημα Galois έχουμε τα παρακάτω αποτελέσματα:

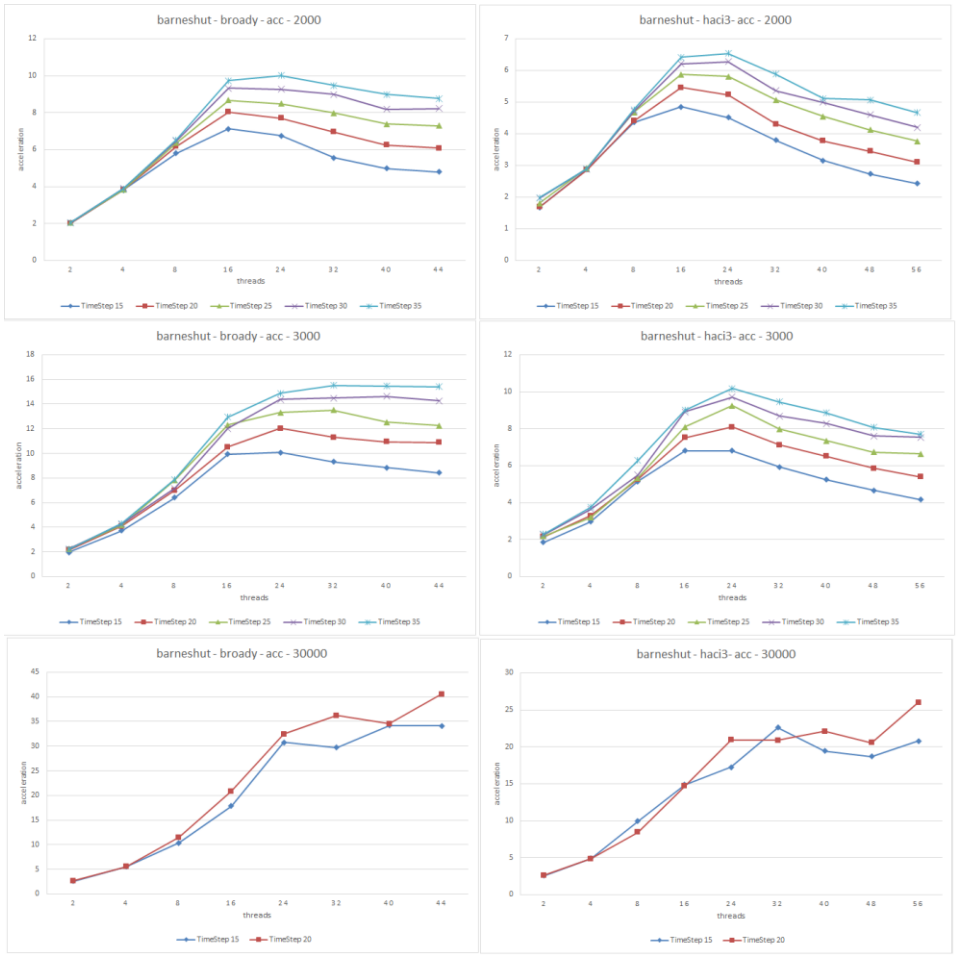

Figure 5: Επιτάχυνση της εκτέλεσης του Galois barneshut αλγορίθμου στην αύξηση των νημάτων.

Από τα παραπάνω φαίνεται ότι η εκτέλεση του αλγορίθμου κλιμακώνει πολύ καλά στην αύξηση των νημάτων. Ο αλγόριθμος στο haci3 δεν φαίνεται να φτάνει την επιτάχυνση που φτάνει στο broady, ωστόσο αυτό συμβαίνει διότι οι εκτελέσεις στο haci3 είναι έως και δύο φορές γρηγορότερες από αυτές στο broady. Ως τάξη μεγέθους, για ένα νήμα, για 25 χρονικά βήματα, το broady κάνει 52 δευτερόλεπτα να τρέξει την εκτέλεση, ενώ το haci3 21. Πέρα από αυτή την διαφορά στην εκτέλεση μεταξύ των μηχανημάτων, ο αλγόριθμος επιδεικνύει εξαιρετική παράλληλη συμπεριφορά. Ωστόσο φαίνεται μια μικρή κάμψη στην αύξηση της επιτάχυνσης όταν τα νήματα αυξάνονται πέρα από τα 28 στο haci3. Αυτό συμβαίνει διότι πέρα από τα 28 νήματα ενεργοποιείται το HyperThreading. Οι παραπάνω πόροι που ζητούνται από τα νήματα για τον ίδιο πυρήνα επηρεάζει αρκετά την εκτέλεση για μικρό αριθμό νημάτων πάνω από τα 28 και ιδιαίτερα για μικρότερα input, ενώ φαίνεται να εξομαλύνεται όταν τα νήματα ανεβαίνουν σημαντικά. Η κάμψη αυτή δεν παρατηρείται στον ίδιο βαθμό στο broady, καθώς στο broady σύστημα κάθε επεξεργαστής έχει 55Mb cache, ενώ στο haci3 35Mb.

5.1.2 TSX

Απενεργοποιώντας το semantic commutativity που χρησιμοποιεί το Galois, τα iterations του Barnes-hut εκτελούνται χωρίς κάποια μέθοδο συγχρονισμού. Αν και οι περισσότερες επαναλήψεις αποτελούνται από iterations που είναι ανεξάρτητα μεταξύ τους, υπάρχουν ορισμένα σημεία στα οποία χρειάζεται. Χρειάζεται συγχρονισμός των iterations στην πρώτη επανάληψη που δημιουργείται το octree, καθώς επίσης και σε ορισμένα σημεία

στην computeForce που περιλαμβάνουν αναγνώσεις και εγγραφές κοινών δεδομένων. Η τελευταία επανάληψη όπου γράφονται οι νέες δυνάμεις δεν έχει καταστάσεις συναγωνισμού.

Η πρώτη προσπάθεια για την υλοποίηση του συγχρονισμού μέσω Transactional Memory, περιλαμβάνει τον ορισμό όλου του iteration ως ένα transaction. Με αυτό τον τρόπο υλοποιείται ένα αρκετά εύκολο coarse grain locking σχήμα, το οποίο όπως υπόσχεται το Transactional Memory, φέρνει καλύτερα αποτελέσματα σε σχέση με coarse grain χρησιμοποιώντας κλειδώματα. Όπως ειπώθηκε και στο κεφάλαιο για το Transactional Memory, ένα transaction προσπαθεί να τελειώσει την εκτέλεση του και να κάνει commit ένα μεταβλητό αριθμό φορών, και αν αποτύχει, το νήμα πηγαίνει σε ένα global lock acquire. Στις παρακάτω εκτελέσεις οι φορές αυτές είναι 25.

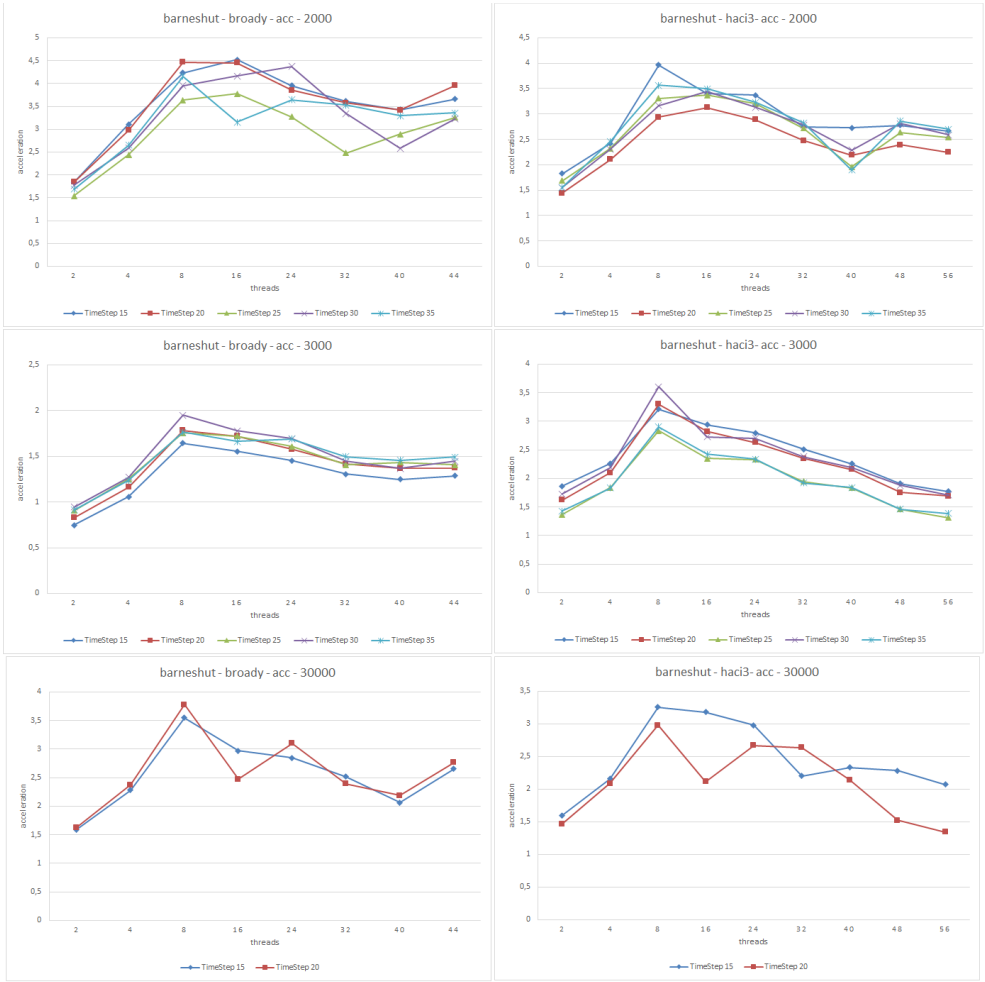

Figure 6: Επιτάχυνση της εκτέλεσης barneshut όταν χρησιμοποιείται TSX σε coarse gain σχήμα

Ο κώδικας στις παραπάνω εκτελέσεις μοιάζει με τον παρακάτω:

```
List bodylist = ...
foreach timestep do {
      Octree octree;
      foreach Body b in bodylist {
            transaction_start();
            octree.Insert(b);
            transaction_end();
      }
      octree.SummarizeSubtrees();
      foreach Body b in bodylist {
            transaction_start();
```

```
                b.ComputeForce(octree);
                transaction_end();
        }
        foreach Body b in bodylist {
                b.Advance();
        }
}
```

Αυτό που φαίνεται ξεκάθαρα είναι ότι τα αποτελέσματα στην Figure 6 δεν μοιάζουν καθόλου με αυτά που έχουμε όταν χρησιμοποιείται ο συγχρονισμός του Galois.

Αυτό το οποίο συμβαίνει στην εκτέλεση φαίνεται καλύτερα μετά την επεξεργασία των στατιστικών που παράγει το TSX κατά την εκτέλεση του.

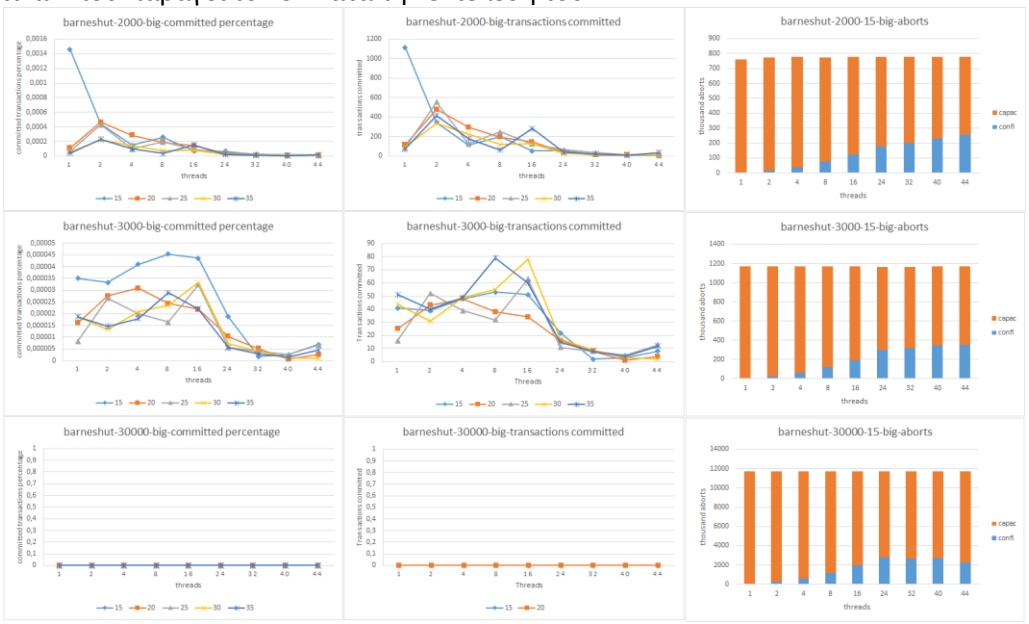

Figure 7: TSX committed και aborted transactions για τον barneshut αλγόριθμο με μεγάλα transactions

Τα στατιστικά στην Figure 7 προκύπτουν όταν κάθε νήμα προσπαθεί να εκτελέσει το transaction 25 φορές πριν πάει στο global lock, όπου στην αριστερή στήλη είναι το ποσοστό των committed transactions προς τις συνολικές προσπάθειες, στην μεσαία στήλη είναι ο αριθμός των committed transaction σε απόλυτο αριθμό, και στην δεξιά στήλη η κατανομή των aborts για timestep 15. Το confl αντιστοιχεί στα conflict aborts και το capac στα capacity aborts. Τα υπόλοιπα είδη abort είναι μηδενικά. Αυτό που φαίνεται ξεκάθαρα είναι ο τεράστιος αριθμός από aborted transactions, ειδικότερα σε μεγαλύτερα input. Ακόμη, στην τελευταία σειρά που αντιστοιχεί στο μεγάλο input, βλέπουμε αυτό που αναφέρεται και από την Intel, ότι δεν δίνεται καμία εγγύηση ότι κάποιο transaction κάποια στιγμή θα εκτελεστεί επιτυχώς. Το κύριο πρόβλημα εδώ είναι τα capacity aborts, και ειδικότερα η επανάληψη του ComputeForce(). Η πρώτη επανάληψη εισαγωγής κόμβων στο octree δεν παράγει πολλά capacity aborts, καθώς το βάθος του δέντρου δεν είναι μεγάλο διότι πρόκειται για δέντρο με 8 παιδιά σε κάθε κόμβο.

Στην επανάληψη του ComputeForce ωστόσο, όλο το δέντρο ξεδιπλώνεται για να υπολογιστούν οι δυνάμεις, και ανάλογα με την ακρίβεια, μπορεί να διαβαστεί και όλο το δέντρο. Αυτό δημιουργεί transactions πολύ μεγάλου μεγέθους τα οποία προκαλούν συνεχώς capacity aborts. Ιδιαίτερα στην τελευταία εκτέλεση, κανένα transaction δεν εκτελέστηκε επιτυχώς, αλλά όλα πήγανε στο global lock. Η αύξηση του αριθμού που θα προσπαθήσει να εκτελεστεί ένα transaction, δεν βοηθάει καθόλου σε αυτήν την περίπτωση, καθώς όσες φορές και να εκτελεστεί ένα transaction μεγάλου μεγέθους, δεν πρόκειται ποτέ να κάνει commit και απλά θα χρονοτριβεί μέχρι να προσπαθήσει να πάρει το global lock.

Για να βελτιωθεί η απόδοση σε αυτόν τον αλγόριθμο, αρκεί να παρατηρήσει κανείς ότι η computeFroce(), δεν κάνει τίποτα άλλο από τα να διαβάζει το δέντρο. Συνεπώς, υλοποιείται μια πιο fine grain λογική, όπου τα `transaction_start()` και `transaction_end()`, δεν μπαίνουν σε όλο το σώμα του iteration, αλλά μόνο στα σημεία που χρειάζεται πραγματικά συγχρονισμός. Συνεπώς σπάμε την δοσοληψίες στα σημεία που δημιουργούνται καταστάσεις συναγωνισμού. Οι κόμβοι και τα δεδομένα του δέντρου δεν θα αλλάξουν σε αυτήν την επανάληψη, αλλά στην επόμενη που εκτελείται η `Advance()` σε κάθε κόμβο.

Σε αυτόν τον αλγόριθμο, δεν χρειάζεται να υλοποιηθεί κάποιο σχήμα για να προστατευθεί το isolation του κάθε iteration, καθώς οι επαναλήψεις είναι τελείως ανεξάρτητες μεταξύ τους. Οποιαδήποτε διαφορετική σειρά εκτέλεσης των δοσοληψιών διαφορετικών iteration δημιουργεί σωστά σειριοποιήσιμα προγράμματα. Σε παρακάτω προβλήματα όμως, όπως τα Delaunay triangulation, και Delaunay Mesh refinement, θα δούμε πως αυτό δεν ισχύει και θα πρέπει να υλοποιηθεί κάποιο σχήμα για να σιγουρευτεί η απομόνωση των iteration.

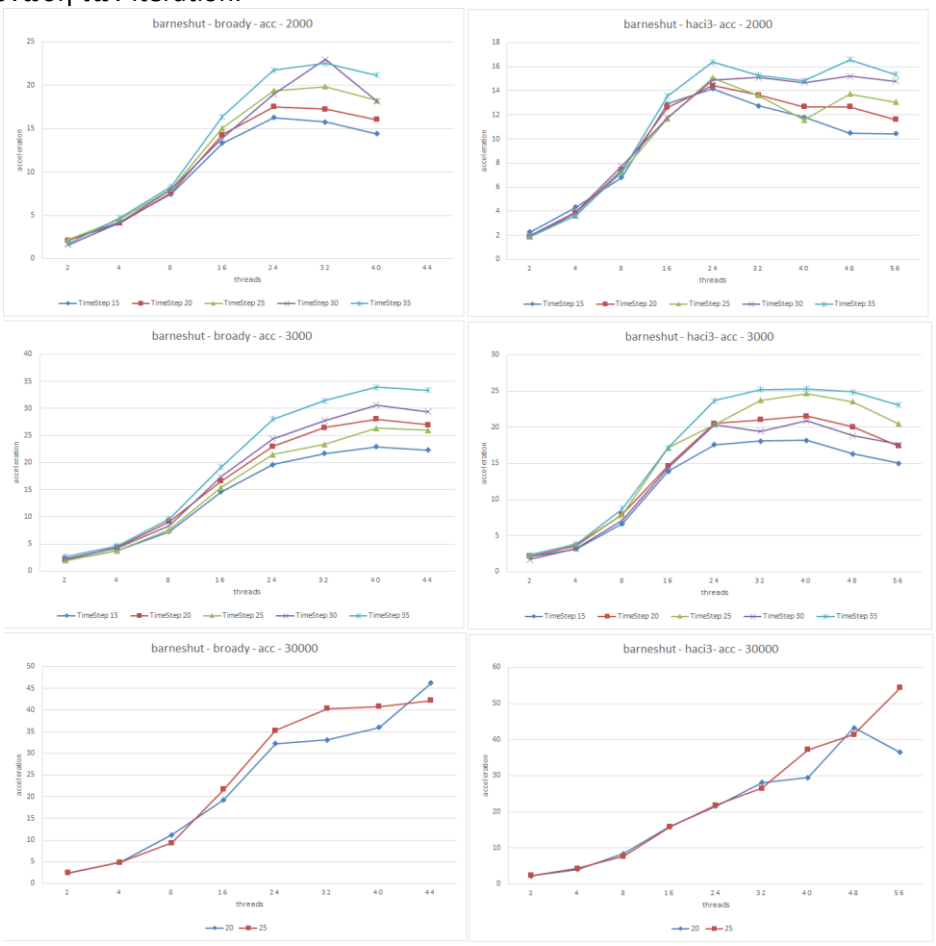

Figure 8: Επιτάχυνση της εκτέλεσης barneshut όταν χρησιμοποιείται TSX σε fine grain σχήμα

Τα αποτελέσματα της εκτέλεσης αυτής είναι πολύ καλύτερα σε σχέση με τα προηγούμενα όπως φαίνεται και στην Figure 8 . Αυτό φαίνεται και στα στοιχεία για τα aborts και commits των transaction.

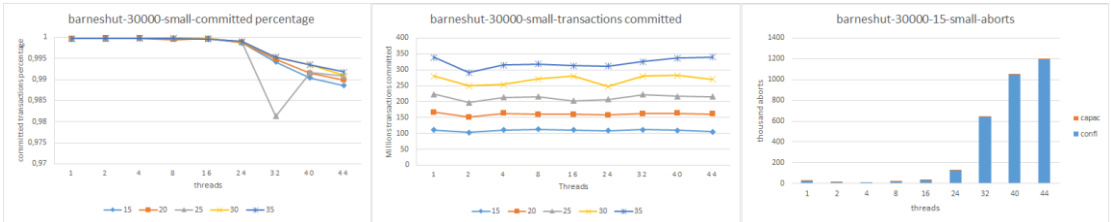

Figure 9:TSX committed και aborted transactions για τον barneshut αλγόριθμο με μικρά transactions

Στην Figure 9 φαίνονται τα στατιστικά της εκτέλεσης με μικρά transactions. Για τα μικρότερα input, η συμπεριφορά είναι ίδια. Τα capacity aborts εδώ είναι σχεδόν μηδενικά, και τα μόνα aborts στο σύστημα είναι τα conflict aborts. Conflict aborts υπάρχουν κάποια ελάχιστα και όταν το σύστημα τρέχει με ένα νήμα, λόγω false sharing, δηλαδή δεδομένα που μπαίνουν στην ίδια cache line μαζί με άλλα που είναι σε transaction mode.

Είναι χαρακτηριστικό ότι η απόδοση στην fine grain tsx υλοποίηση είναι καλύτερη από αυτήν του Galois. Αυτό συμβαίνει διότι ο συγχρονισμός του Galois υλοποιείται σε όλο το iteration. Το Galois σύστημα δεν μπορεί να γνωρίζει τι πραγματικά χρειάζεται έλεγχο και τι όχι. Είναι υλοποιημένο για να εκτελεί τα iteration φροντίζοντας να τηρούνται όλες οι ACID ιδιότητες. Αυτό προκαλεί στα νήματα και στο Runtime σύστημα να βάζει χρόνο εκτέλεσης σε ελέγχους που δεν χρειάζονται. Σε αντίθεση, ο προγραμματιστής του αλγορίθμου ξέρει τι μπορεί να εκτελεστεί παράλληλα και τι όχι και μπορεί να εφαρμόσει τους ελέγχους μόνο εκεί που χρειάζονται.

5.2 Clustering

Clustering αλγόριθμοι χρησιμοποιούνται για την ομαδοποίηση αντικειμένων σε μια κοινή δομή, σύμφωνα με κάποιο χαρακτηριστικό τους. Υπάρχουν δύο τύποι αλγορίθμων: Agglomerative clustering και Divisive clustering. Η Agglomerative λογική υλοποιεί μια από κάτω προς τα πάνω τεχνική, όπου κάθε αντικείμενο είναι μια δική του ομάδα, και σε κάθε βήμα του αλγορίθμου επιλέγονται δύο ομάδες και ενώνονται. Αντιθέτως, η Divisive τεχνική, ξεκινάει με μια μεγάλη ομάδα των αντικειμένων, και στην πορεία του αλγόριθμου η ομάδα αυτή σπάει σε μικρότερες. Το αποτέλεσμα ενός clustering αλγορίθμου είναι ένα dendrogram, μια ιεραρχική δεντρική δομή που χρησιμοποιείται για αναπαράσταση ομαδοποιημένων αντικειμένων ανά δυάδες.

Και οι δύο αυτές τεχνικές έχουν ένα ίδιο χαρακτηριστικό. Ο αλγόριθμος που εφαρμόζεται για την ένωση δύο ομάδων ή το σπάσιμο της ομάδας σε δύο μικρότερες. Και στις δύο περιπτώσεις εφαρμόζεται άπληστος αλγόριθμος που καθιστά την πολυπλοκότητα του clustering σε $O(n^2 log(n))$.

Η είσοδος του αλγορίθμου είναι δεδομένα κόμβων, σε ένα ν-διάσταστο χώρο στην γενική περίπτωση, και ένα μέτρο σύγκρισης της ομοιότητας δύο τέτοιων κόμβων. Στην πιο απλή περίπτωση, η ομοιότητα των κόμβων δεν είναι κάτι άλλο από την απόσταση τους στον χώρο. Γι' αυτό το λόγο, παρακάτω θα χρησιμοποιηθεί αυτή η μετρική για την περιγραφή του αλγορίθμου.

Στην Agglomerative τεχνική που υλοποιείται στο Galois, η ομαδοποίηση δύο κόμβων γίνεται όταν και για τους δύο κόμβους, ο υπολογισμός του κοντινότερου κόμβου δίνει τον άλλον κόμβο. Όταν δηλαδή δύο κόμβοι συμφωνούν ότι είναι πιο κοντά μεταξύ τους από κάθε άλλο κόμβο. Παρακάτω φαίνεται ψευδοκώδικας από την υλοποίηση του αλγορίθμου σε λογική worklist, όπως είναι και στο Galois:

```
worklist = new Set(input_points);
kdtree = new KDTree(input_points);
for each Element p in worklist do {
       if (/* p already clustered */) continue;
       q = kdtree.findNearest(p);
       if (q == null) break;
       r = kdtree.findNearest(q);
       if (p == r) {
              Element e = cluster(p,q);
              kdtree.remove(p);
              kdtree.remove(q);
              kdtree.add(e);
              worklist.add(e);
       } else {
              worklist.add(p); //add back to worklist
```

```
        }
}
```

Στην πρώτη γραμμή, όλοι οι αρχικοί κόμβοι εισάγονται στην δουλειά εργασιών. Στην συνέχεια, από αυτούς τους κόμβους χτίζεται ένα *k-d tree*. Η δομή αυτή είναι χρήσιμή γιατί κρατάει τους κόμβους με έναν τρόπο όπου μπορείς γρήγορα να βρεις τον κοντινότερο γείτονα. Μοιάζει αρκετά με ένα *octree*, καθώς το *k-d tree* αποτελεί δεντρική υλοποίηση για ν-διάστατο χώρο αντικειμένων. Στην συνέχεια αφού έχει κατασκευαστεί η δομή αυτή, ο αλγόριθμος πηγαίνει σε μια επανάληψη πάνω στα στοιχεία της worklist. Για κάθε κόμβο στην δουλειά εργασιών, βρίσκεται ο κοντινότερος γείτονας του κόμβου, και αν αυτός δεν υπάρχει τότε ο αλγόριθμος έχει τερματίσει. Αν υπάρχει, τότε βρίσκεται ο κοντινότερος γείτονας του κόμβου αυτού, και αν αυτός ο κόμβος είναι ίδιος με τον αρχικό τότε μπορεί να γίνει ομαδοποίηση. Αν δεν είναι τότε θα πρέπει να μπει πάλι στην worklist ο κόμβος, καθώς θα πρέπει να ομαδοποιηθεί σε επόμενο βήμα.

Αν οι δύο κόμβοι είναι ίδιοι τότε δημιουργείται μια ομάδα με τα χαρακτηριστικά τους, αφαιρούνται από το k-d tree οι παλαιοί κόμβοι, και ο νέος κόμβος εισάγεται στο k-d tree και στην λίστα εργασιών.

Clustering αλγόριθμοι χρησιμοποιούνται σε Data mining και Machine Learning εφαρμογές.

5.2.1 Galois

Ο παραλληλισμός σε αυτόν τον αλγόριθμο, προκύπτει από το γεγονός ότι διαφορετικοί κόμβοι, που δεν είναι κοντά μεταξύ τους, μπορούν να επεξεργαστούν με ανεξάρτητο τρόπο. Αυτό σημαίνει ότι η επανάληψη πάνω στην worklist μπορεί να εκτελεστεί με παράλληλο τρόπο. Για να μην υπάρχουν εξαρτήσεις κατά την επεξεργασία ενός κόμβου από ένα νήμα, πρέπει οι δύο κόμβοι προς ομαδοποίηση να μην επεξεργάζονται από κανένα άλλο νήμα, και ο νέος κόμβος που θα δημιουργηθεί να μην προκαλεί εξαρτήσεις στον υπολογισμό της απόστασης με κανέναν άλλο κόμβο που επεξεργάζεται εκείνη την στιγμή.

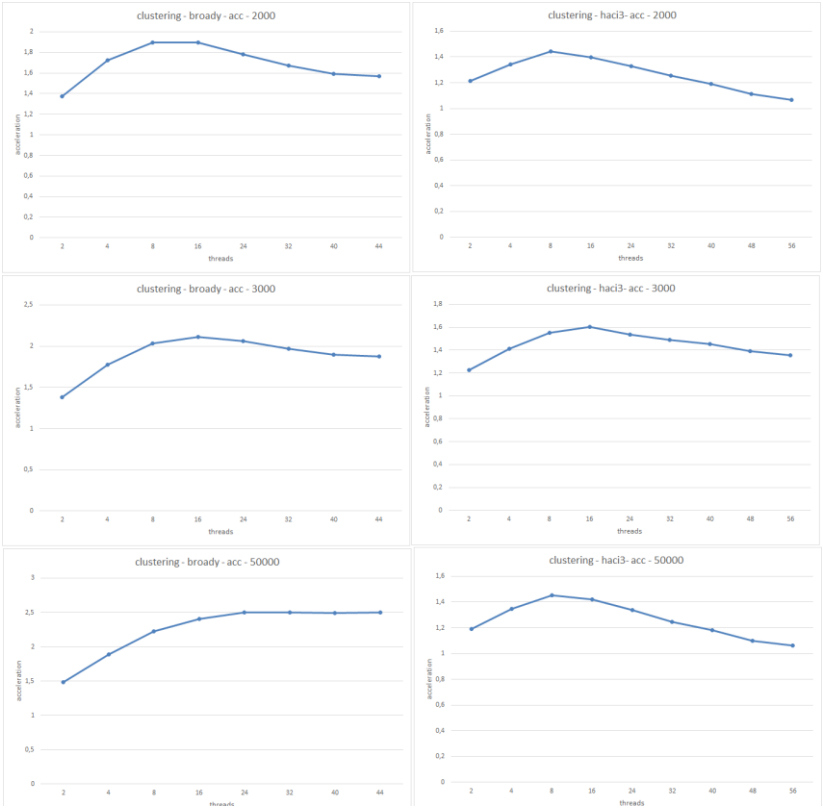

Figure 10: Επιτάχυνση της εκτέλεσης του Galois clustering αλγορίθμου στην αύξηση των νημάτων

Οι κόμβοι που υπάρχουν στην λίστα εργασιών δεν έχουν συγκεκριμένη σειρά με την οποία πρέπει να εκτελεστούν. Αυτό σημαίνει ότι η επανάληψη στο Galois γίνεται με unordered set iterator.

Τα αποτελέσματα στην Figure 10 δεν δείχνουν καλή παράλληλη συμπεριφορά. Το κύριο πρόβλημα που έχει αυτός ο αλγόριθμος είναι ότι ο διαθέσιμος παραλληλισμός φθίνει, όσο ομαδοποιούνται περισσότερα στοιχεία. Τα παραπάνω νήματα για την επεξεργασία κόμβων δεν επιδρούν θετικά στην απόδοση καθώς όσο μικραίνει ο αριθμός των κόμβων προς επεξεργασία, τόσο περισσότερες εξαρτήσεις υπάρχουν. Χαρακτηριστικό είναι ότι η απόδοση στον μέγιστο αριθμό νημάτων, στο haci3 μηχάνημα, είναι αρκετά κοντά σε αυτόν με δύο μόλις νήματα.

5.2.2 TSX

Όπως και στην περίπτωση του Barnes-hut έτσι και εδώ, η πρώτη προσέγγιση είναι όλος ο κώδικας της επανάληψης να είναι ένα μεγάλο transaction. Ο αλγόριθμος του clustering έχει αρκετά παραπάνω σημεία στα οποία τα νήματα χρειάζονται συγχρονισμό από τον barneshut, καθώς τώρα υπάρχει κοινή δομή η οποία γράφεται και διαβάζεται ταυτόχρονα από διαφορετικά νήματα. Συγκεκριμένα η *findNearest(p)* κλήση, διαβάζει δεδομένα από το δέντρο, ενώ η *remove(p)* και *add(e)* διαβάζουν και γράφουν πάνω στο k-d tree. Εδώ υπάρχει και η *add* ενός καινούριου κόμβου στην worklist, η οποία όμως worklist δεν χρειάζεται συγχρονισμό, καθώς οι κλήσεις σε αυτήν είναι μικρές *push* και *pop* λειτουργίες που προστατεύονται από κλείδωμα. Ωστόσο, εισάγοντας και αυτό το κομμάτι κώδικα μέσα στο transaction, νήματα που προσπαθούν να πάρουν το lock προκαλούν εξαρτήσεις με νήματα που ήδη έχουν το lock. Ορίζοντας ως αριθμό φορών που θα προσπαθήσει να εκτελέσει ένα transaction το TSX τις 25, έχουμε τα παρακάτω αποτελέσματα:

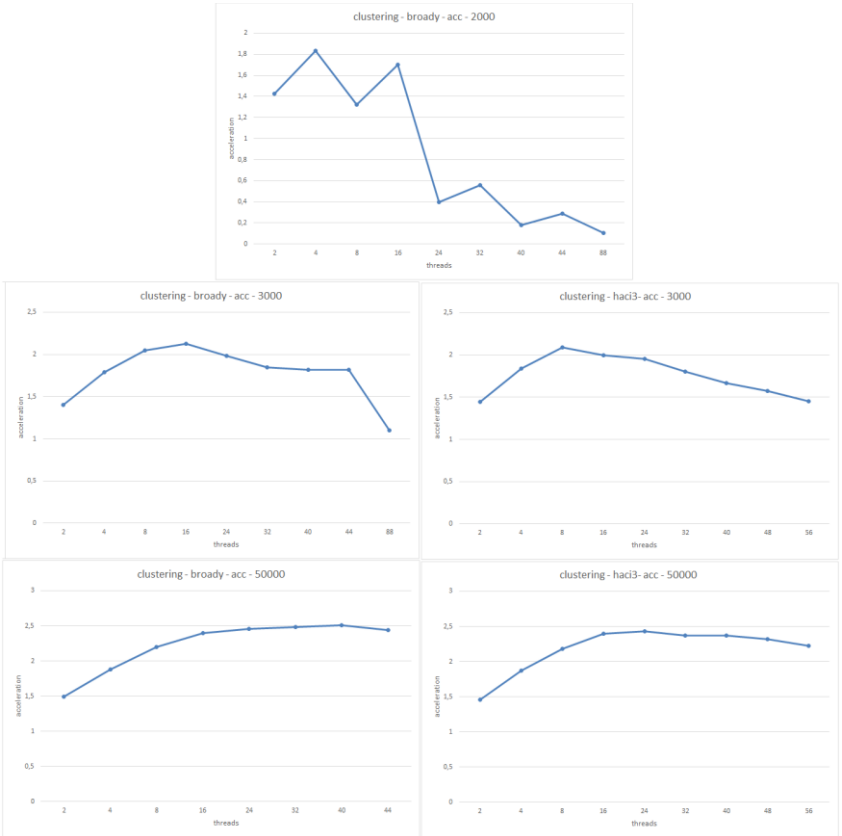

Figure 11: Επιτάχυνση της εκτέλεσης clustering όταν χρησιμοποιείται TSX σε coarse grain σχήμα

Από την Figure 11 φαίνεται ότι η επιτάχυνση της εκτέλεσης είναι ίδια με αυτήν του Galois. Όπως συζητήθηκε και προηγουμένως, ο αλγόριθμος αυτός δεν έχει πολλά σημεία παραλληλοποίησης, καθώς σε κάθε βήμα εκτέλεσης μειώνεται ο χώρος των κόμβων. Ωστόσο, για να καταλάβουμε καλύτερα τον αλγόριθμο ας κοιτάξουμε τα στατιστικά που παράγει το TSX:

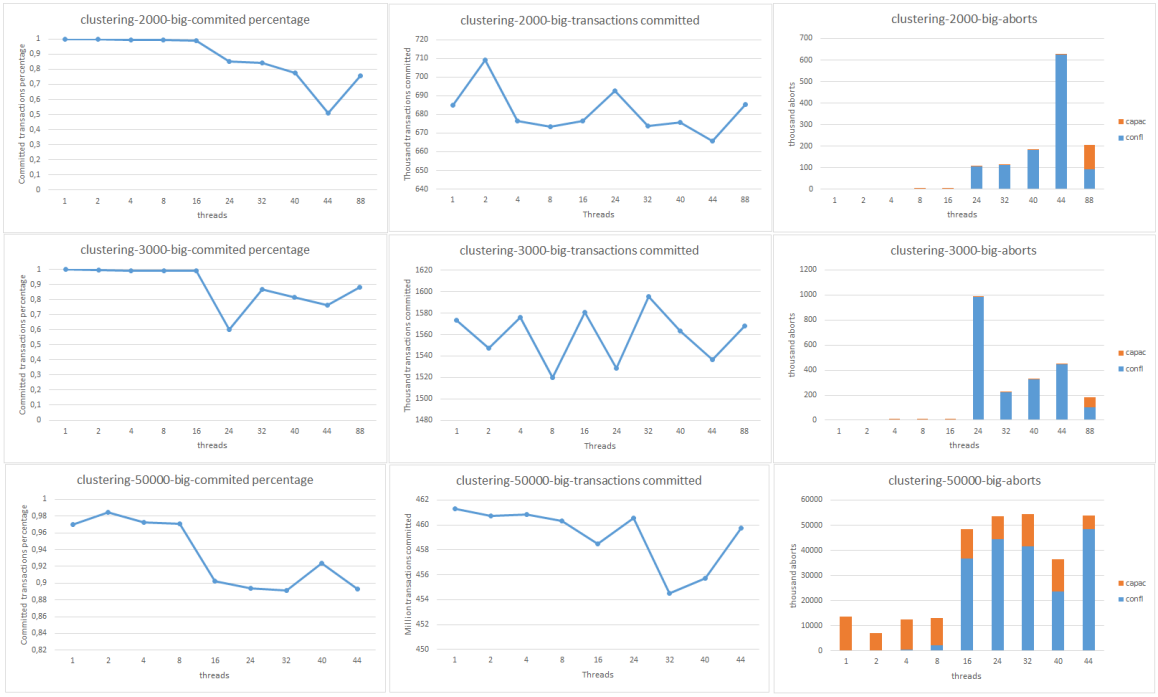

Figure 12:TSX committed και aborted transactions για τον clustering αλγόριθμο με μεγάλα transactions

Τα στατιστικά των transaction είναι πολύ καλύτερα σε σχέση με αυτά του barneshut. Η διαφορά είναι ότι στον barneshut αλγόριθμο, για να υπολογιστεί η δύναμη ενός κόμβου πρέπει να συνυπολογιστούν οι δυνάμεις που ασκούν όλοι οι άλλοι κόμβοι, δηλαδή να διαβαστεί ένα μεγάλος μέρος του δέντρου. Αυτό μεγαλώνει την μνήμη που χρειάζεται ένα transaction. Αντιθέτως, στον clustering αλγόριθμο, η ιεραρχική k-d tree δομή χρειάζεται να βρει μόνο τον κοντινότερο γείτονα. Αυτό σημαίνει διάβασμα ενός μικρού μέρους του δέντρου, κοντά στον κόμβο προς εξέταση. Στον Barnes-hut αλγόριθμο, το πρόβλημα των capacity aborts ήταν τόσο μεγάλο, που τα transactions που κάναν commit επιτυχώς τείνανε προς το μηδέν, καθώς μεγάλωνε η είσοδος. Ωστόσο, και ο clustering αλγόριθμος υποφέρει από προβλήματα μνήμης, καθώς στο μεγαλύτερο input υπάρχει ένα σταθερός αριθμός από capacity aborts. Όπως αναμένεται, ο αριθμός αυτός είναι περίπου σταθερός, ενώ τα conflict aborts αυξάνονται καθώς μεγαλώνει ο αριθμός των νημάτων.

Ωστόσο, υπάρχει χώρος προς βελτίωση, καθώς ο κώδικας που υπάρχει μέσα στην επανάληψη, δεν χρειάζεται να βρίσκεται όλος σε ένα μεγάλο transaction. Αρκεί να παρατηρήσουμε ότι, έστω ένας κόμβος a που βρίσκει τον κόμβο b ως κοντινότερο, και ο κόμβος b βρίσκει τον κόμβο a ως κοντινότερο. Αυτοί οι δύο κόμβοι θα ομαδοποιηθούν. Αν υπάρχει ένα άλλο νήμα που τρέχει ταυτόχρονα για τον κόμβο c, που βρίσκει ότι έχει τον b κοντινότερο κόμβο, δεν θα τους ομαδοποιήσει ποτέ καθώς ο b έχει τον a κόμβο κοντινότερα. Αυτό σημαίνει ότι το δεύτερο νήμα θα βάλει τον c πάλι στην worklist. Ακόμη, όταν δύο κόμβοι συμφωνούν ότι βρίσκονται κοντινότερα μεταξύ τους, μπορούν να ομαδοποιηθούν με ασφάλεια, καθώς οποιοσδήποτε καινούριος κόμβος εισαχθεί, αποκλείεται να βρίσκεται τόσο κοντά που να τους επηρεάζει στον υπολογισμό της απόστασης. Αυτοί οι δύο λόγοι σημαίνουν ότι απλά πρέπει να προστατευθεί η πρόσβαση των νημάτων στα κοινά δεδομένα, χωρίς να υλοποιηθεί κάποιο ακόμα σχήμα σειριοποίησης των transactions. Αυτό δεν ισχύει για τα δύο επόμενα benchmarks που θα χρησιμοποιηθούν:

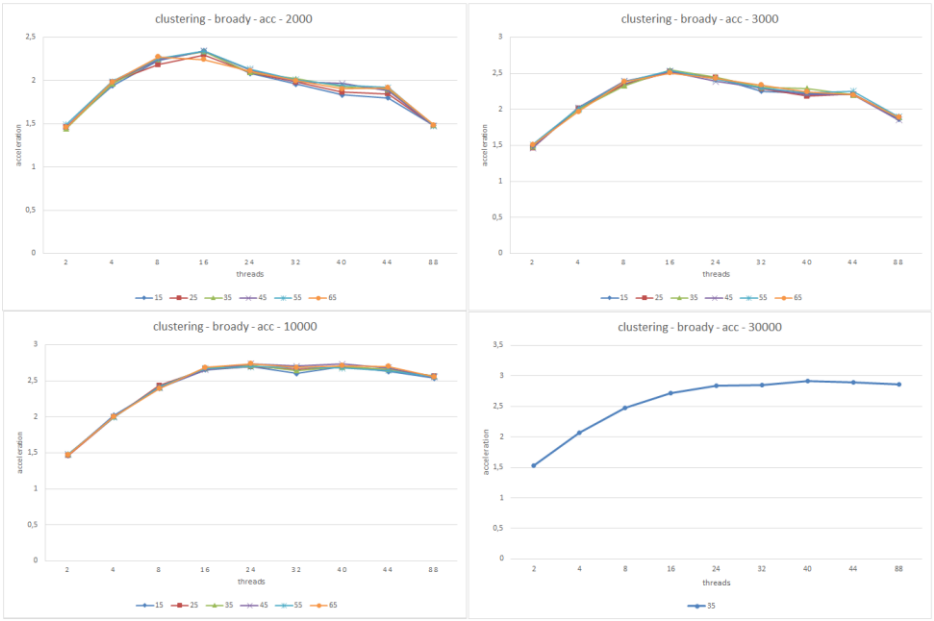

Figure 13: Επιτάχυνση της εκτέλεσης clustering όταν χρησιμοποιείται TSX σε fine grain σχήμα

Τα διαφορετικά γραφήματα, αντιπροσωπεύουν διαφορετικές εκτελέσεις, όταν ο αριθμός των φορών που θα προσπαθήσει να τρέξει ένα transaction μεταβάλλεται από 15 μέχρι και 65. Παρατηρούμε ωστόσο, ότι ο αριθμός αυτός δεν έχει κανένα απολύτως αντίκτυπο στην εκτέλεση. Πέρα από αυτό, τα αποτελέσματα είναι παρόμοια με αυτά του Galois. Αν και η επιτάχυνση είναι ελάχιστα μεγαλύτερη στο μέγιστο σημείο της κάθε εκτέλεσης σε σχέση με το Galois, δεν φτάνει σε καμία περίπτωση τα επίπεδα του barneshut. Για μικρά input παρατηρείται μια κάμψη στην επιτάχυνση, καθώς ο διαθέσιμος παραλληλισμός φθίνει αρκετά γρήγορα. Ας δούμε τα αποτελέσματα των στατιστικών του TSX:

Figure 14: TSX committed και aborted transactions για τον clustering αλγόριθμο με μικρά transactions

Ο αριθμός των επιτυχώς ολοκληρωμένων transactions παρουσιάζει σταθερή συμπεριφορά. Από την Figure 14 παρατηρούμε ότι αν και μειώθηκε ο αριθμός των capacity aborts στις εκτελέσεις με μικρό αριθμό νημάτων, το πρόβλημα της υπερχείλισης μνήμης είναι ορατό ιδιαίτερα στην εκτέλεση των 88 νημάτων, όπου υπάρχει μια σημαντική μείωση των διαθέσιμων πόρων λόγω ενεργοποίησης της τεχνολογίας HyperThreading. Πέρα από αυτό, ο

αριθμός των συνολικών aborts είναι περίπου διπλάσιος, σε απόλυτο αριθμό, σε σχέση με τις εκτελέσεις με μεγάλα transactions. Αυτό οφείλεται κατά κύριο λόγω στα conflict aborts. Επίσης υπάρχουν και πάλι conflict aborts σε εκτελέσεις με ένα νήμα λόγω false sharing, αλλά είναι τόσο λίγα που επηρεάζουν ελάχιστα την εκτέλεση.

Το γεγονός ότι η εκτέλεση με TSX έχει ελαφρώς καλύτερη επιτάχυνση από αυτήν του Galois, οφείλεται στο γεγονός ότι δεν τρέχει από πίσω το κομμάτι του ελέγχου για εξαρτήσεις, καθώς το κομμάτι αυτό το έχει αναλάβει το TSX.

5.3 Delaunay triangulation

Ο αλγόριθμος Delaunay triangulation χρησιμοποιείται για την δημιουργία τριγώνων μεταξύ σημείων πάνω σε έναν επίπεδο, έτσι ώστε κάθε περιγεγραμμένος κύκλος ενός τριγώνου, να περιέχει έχει ως εσωτερικά σημεία μόνο τα τρία σημεία από τα οποία περνάει, τα οποία είναι και οι κορυφές του τριγώνου.

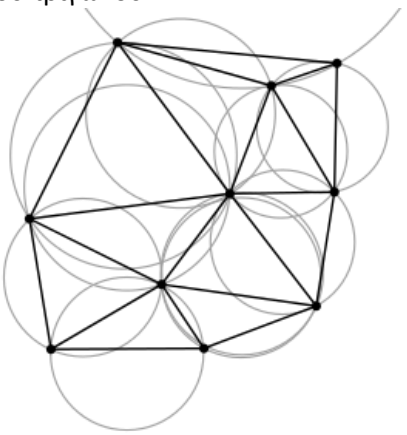

Figure 15: Παράδειγμα Delaunay triangulation

Ο αλγόριθμος έχει ως είσοδο ένα σύνολο σημείων στο δισδιάστατο επίπεδο, και δίνει ως έξοδο ένα σύνολο ακμών μεταξύ των σημείων. Στην περίπτωση όπου τα σημεία της εισόδου βρίσκονται όλα πάνω σε μια γραμμή, τότε δεν υπάρχει τριγωνοποίηση. Αν υπάρχουν τέσσερα ή παραπάνω σημεία που βρίσκονται πάνω στον ίδιο κύκλο, τότε η τριγωνοποίηση δεν είναι μοναδική. Στον εξεταζόμενο αλγόριθμο εξετάζονται σημεία και τρίγωνα πάνω σε δισδιάστατο επίπεδο. Αν ωστόσο ο περιγεγραμμένος κύκλος αναλογιστεί ως περιγεγραμμένη σφαίρα, τότε η ίδια λογική μπορεί να αναχθεί και σε περισσότερες διαστάσεις.

Οι διάφορες τεχνικές που χρησιμοποιούνται για να υλοποιηθεί ο αλγόριθμος είναι οι Flip, Incremental και Divide and conquer.

Η Flipping λογική προκύπτει από την παρακάτω ιδιότητα:

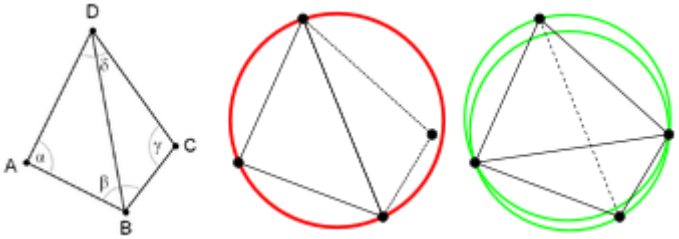

Figure 16: Delaunay triangulation μετατροπή τριγώνων

Στην Figure 16 αν οι γωνίες α και γ των δύο τριγώνων που έχουν μια κοινή ακμή, είναι αθροιστικά μικρότερες από 180^0 τότε τα τρίγωνα ικανοποιούν την ιδιότητα της τριγωνοποίησης (ότι οι περιγεγραμμένοι κύκλοι περιέχουν μόνο τις κορυφές του τριγώνου). Αν όμως οι γωνίες είναι αθροιστικά μεγαλύτερες από 180 μοίρες, η ιδιότητα δεν

ικανοποιείται. Σε αυτήν την περίπτωση, αρκεί να παρατηρήσει κανείς ότι η αλλαγή της φοράς της κοινής ακμής ώστε να διέρχεται από τα άλλα δυο σημεία, δημιουργεί τρίγωνα και γωνίες που αποτελούν ορθή τριγωνοποίηση. Αυτή η ιδιότητα γενικεύεται και σε περισσότερες διαστάσεις.

Έχοντας υπόψη αυτήν την τεχνική για την δημιουργία ορθών τριγώνων, οι Flip αλγόριθμοι δημιουργούν μια τυχαία τριγωνοποίηση των αρχικών σημείων και στην συνέχεια γίνεται flipping κοινών ακμών από μη ορθά τρίγωνα, μέχρι αυτά να εξαλειφθούν. Η τεχνική αυτή έχει τετραγωνική πολυπλοκότητα και μπορεί να γενικευθεί και σε περισσότερες διαστάσεις.

Μια διαφορετική προσέγγιση είναι η Incremental. Κάθε φορά εισάγεται και ένας νέος κόμβος από το αρχικό σύνολο, και το τρίγωνο, μέσα στο οποίο περιέχεται αυτός ο κόμβος, χωρίζεται σε τρία τρίγωνα που έχουν όλα ως κοινό κόμβο το νεοεισαχθέν σημείο. Αν για κάποιο από αυτά τα τρίγωνα δεν ικανοποιείται η ιδιότητα της τριγωνοποίησης, τότε εφαρμόζεται η Flipping τεχνική. Η υλοποίηση της λογικής αυτής είναι γραμμική ως προς την είσοδο, αλλά για κάθε καινούριο κόμβο που εισέρχεται πρέπει να βρεθεί το τρίγωνο που το περιέχει. Αυτό μπορεί να είναι και γραμμικό ως προς τον αριθμό των τριγώνων, οδηγώντας και πάλι σε $O(n^2)$. Ωστόσο, αν κάθε τρίγωνο κρατάει αναφορές στα τρίγωνα που το αντικαταστήσανε, τότε αρχίζοντας από ένα αρχικό μεγάλο τρίγωνο, η αναζήτηση έχει λογαριθμική πολυπλοκότητα καθιστώντας τον Incremental αλγόριθμο O(nlogn).

Μια αρκετά διαφορετική προσέγγιση είναι η διαίρει και βασίλευε. Το αρχικό σύνολο των κόμβων χωρίζεται σε δύο υποσύνολά, τα δύο υποσύνολα τριγωνοποιούνται αναδρομικά και στην συνέχεια ενώνονται σε ένα. Η ένωση των δύο συνόλων απαιτεί γραμμική πολυπλοκότητα, δίνοντας τελικά O(nlogn). Η τεχνική αυτή είναι η γρηγορότερη από τις δύο προηγούμενες.

Η υλοποίηση που θα χρησιμοποιηθεί είναι βασισμένη στην Incremental λογική. Επιπρόσθετα, κάθε κόμβος που έχει εισαχθεί με επιτυχία στον γράφο εισάγεται και σε ένα τετραδικό δέντρο, που χρησιμοποιείται για να βρίσκεται ο κοντινότερος γειτονικός κόμβος κάθε σημείου. Κάθε κόμβος κρατάει επίσης μια αναφορά σε ένα τυχαίο τρίγωνο από αυτά στα οποία συμμετέχει. Με αυτόν τον τρόπο, κάθε φορά διαλέγεται ένα κόμβος p από την worklist, βρίσκεται ο κοντινότερος του κόμβος q μέσα στον γράφο (O(logn) λόγω του δέντρου), βρίσκεται το τρίγωνο t που περιβάλλει τον p και ελέγχονται όλα τα γειτονικά τρίγωνα του t για το αν ο περιγεγραμμένος κύκλος τους περιλαμβάνει το p. Αυτά τα τρίγωνα που θα συλλεχθούν αποτελούν την κοιλότητα προς εξέταση του κόμβου p, και θα επεξεργαστούν για να γίνει ορθή τριγωνοποίηση. Όταν τελειώσει η επεξεργασία ο νέος κόμβος εισάγεται στο τετραδικό δέντρο. Παρακάτω φαίνεται αυτή λογική:

```
Set<Point> points = /* read points to insert */;
Mesh m = new Mesh();
Triangle large = new Triangle(points);
m.add(large);
Workset ws = new Workset();
ws.addAll(points);
QuadTree tree = new QuadTree();
foreach (Point p : ws) {
      Point q = tree.find(p)
      Triangle t = findContainingTriangle(p, q);
      Cavity c = new Cavity(t, p);
      c.expand();
      c.retrianglate();
      m.updateMesh(c);
      if (*)
      tree.update(...)
}
```

Ο αλγόριθμος ξεκινάει με ένα μεγάλο τρίγωνο, και όπως και στα προηγούμενα παραδείγματα, οι κόμβοι προς επεξεργασία εξάγονται από την worklist και επεξεργάζονται.

Έχοντας βρει ένα τρίγωνο που περιέχει το p, η επεξεργασία χωρίζεται σε τρία βήματα. Πρώτο βήμα είναι η `expand()`, που βρίσκει όλα τα τρίγωνα που περιέχουν το p, γίνεται η τριγωνοποίηση με την `retriangulate()` και στην συνέχεια γράφονται οι αλλαγές πάνω στον γράφο `updateMesh()`. Οι `expand()` και `retriangulate()` δεν γράφουν στα κοινά δεδομένα. Αυτό είναι εξαιρετικά σημαντικό για τον συγχρονισμό, καθώς οι μόνες κλήσεις που γράφουν στα κοινά δεδομένα είναι η `updateMesh()` και η `tree.update()`.

5.3.1 Galois

Οι κύριες δομές δεδομένων που χρησιμοποιούνται στην υλοποίηση του Galois, είναι i) η thread safe worklist, ii) ένας `unordered set iterator` για την εξαγωγή κόμβων από την λίστα εργασιών, καθώς δεν έχει σημασία η σειρά με την οποία θα επεξεργαστούν τα σημεία παρά μόνο ο τελικός γράφος να είναι ορθός, iii) ο γράφος m όπου τα τρίγωνα αναπαρίστανται ως κόμβοι και οι ακμές είναι μεταξύ τριγώνων στα οποία υπάρχουν κοινές πλευρές, ii) και το τετραδικό δέντρο.

Για κάθε κόμβο προς επεξεργασία και εισαγωγή στον γράφο, βρίσκεται μια γειτονιά τριγώνων κοντά σε αυτόν, και κάποια από αυτά τα τρίγωνα θα διαφοροποιηθούν για να περιλαμβάνουν και τον καινούριο κόμβο. Αυτή η γειτονιά γύρω από τον κόμβο ονομάζεται κοιλότητα, ή αλλιώς Cavity μέσα στον κώδικα. Δεδομένου ότι οι κόμβοι βρίσκονται αρκετά μακριά ο ένας με τον άλλον, καθώς υπάρχει μεγάλος αριθμός σημείων, είναι εξαιρετικά πιθανό οι κοιλότητες αρκετών κόμβων να μην έχουν εξαρτήσεις μεταξύ τους. Αυτό σημαίνει ότι μπορούν να γίνουν εισαγωγές στον γράφο ταυτόχρονα για διάφορους κόμβους. Στην προς εξέταση υλοποίηση χρησιμοποιείται Incremental λογική. Αυτό σημαίνει ότι ο διαθέσιμος παραλληλισμός στην αρχή του αλγορίθμου είναι μικρός, καθώς ξεκινάει από ένα μόλις τρίγωνο και στην συνέχεια μεγαλώνει μέχρι να τελειώσει το input.

Figure 17: Επιτάχυνση της εκτέλεσης του Galois del.Ttriangulation στην αύξηση των νημάτων

Στην Figure 17 παρατηρούμε ότι στα μικρά input η επιτάχυνση της εκτέλεσης είναι πάρα πολύ μικρή, και συνεχίζει να μικραίνει στην αύξηση των νημάτων. Σε μεγαλύτερα input ωστόσο, φαίνεται μεγαλύτερος διαθέσιμος παραλληλισμός με τον χρόνο εκτέλεσης να είναι έως και 12 φορές μικρότερος. Αυτό επιβεβαιώνει ότι ο αλγόριθμος για να δείξει καλή παράλληλη συμπεριφορά πρέπει να έχει ένα αρκετά μεγάλο αριθμό από αρχικά δεδομένα. Παρατηρείται επίσης και μια κάμψη της εκτέλεσης στο μεγαλύτερο input όταν τα νήματα αυξάνονται σε 32, για να αυξηθεί και πάλι μόνο στον μέγιστο δυνατό αριθμό νημάτων. Πάνω από τα 28 νήματα στο haci3 μηχάνημα, ενεργοποιείται το HyperThreading, το οποίο περιορίζει σημαντικά τους διαθέσιμους πόρους σε κάθε νήμα.

5.3.2 TSX

Όπως και στα προηγούμενα παραδείγματα, η πρώτη προσπάθεια για αντικατάσταση του semantic commutativity και χρήση Transactional Memory αποτελεί ο ορισμός όλου του κώδικα ενός iteration ως μια μεγάλη δοσοληψία. Σε αυτόν τον αλγόριθμο, τα σημεία που χρειάζεται συγχρονισμός των νημάτων είναι οι δύο κλήσεις `updateMesh()` και `tree.update()`. Αυτές είναι οι μόνες κλήσεις που γράφουν στα κοινά δεδομένα. Ωστόσο, οι προηγούμενες κλήσεις εύρεσης κόμβων και τριγώνων διαβάζουν από τα δεδομένα, συνεπώς θα ήταν λάθος να βγουν έξω από τον κώδικα της δοσοληψίας. Αξίζει να σημειωθεί εδώ ότι η διαφορετική σειρά εξαγωγής κόμβων από την worklist, ή διαφορετική σειρά επεξεργασίας των κόμβων από κάποιο νήμα, οδηγεί σε διαφορετικό τελικό γράφο, που όμως αποτελεί σωστή τριγωνοποίηση. Στις παρακάτω εκτελέσεις ο αριθμός των φορών που θα προσπαθήσει το transaction να εκτελεστεί είναι 25.

Figure 18: Επιτάχυνση της εκτέλεσης στον del. Triangulation με TSX σε coarse grain σχήμα

Από την Figure 18 βλέπουμε ότι τα αποτελέσματα είναι χειρότερα σε σχέση με αυτά του Galois. Υπάρχουν αρκετά περιθώρια βελτίωσης με την συρρίκνωση των transactions. Σε αυτόν τον αλγόριθμο ωστόσο, ο αριθμός των capacity aborts δεν φαίνεται να είναι τόσο μεγάλο πρόβλημα.

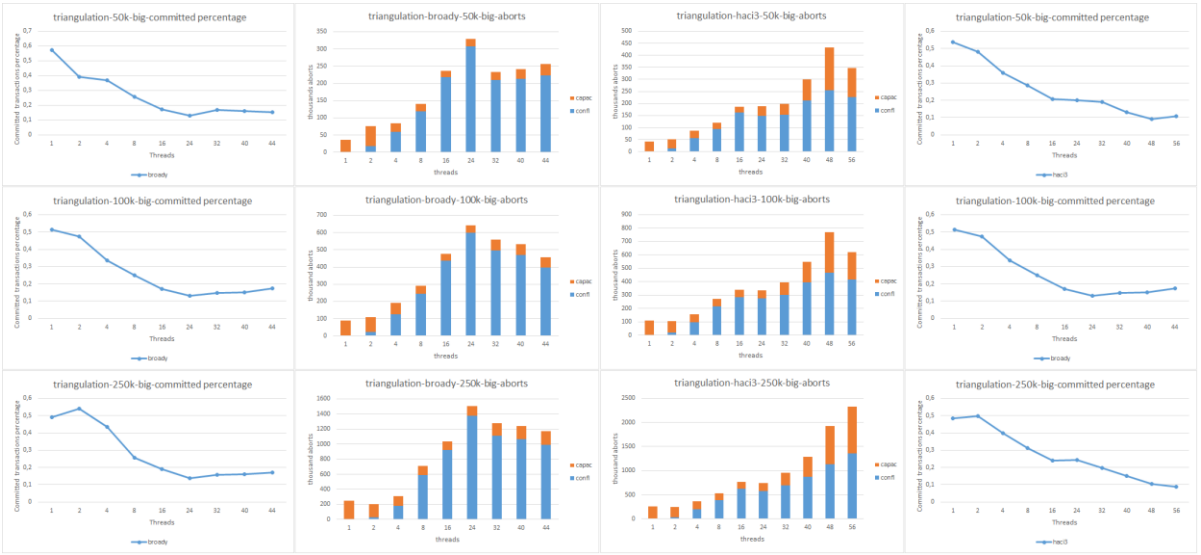

Figure 19: Committed και aborted transactions για del. Triangulation με TSX coarse grain σχήμα

Στην Figure 19 βλέπουμε ότι τα capacity aborts δεν αποτελούν τόσο μεγάλο πρόβλημα όσο τα conflict aborts. Αυτό σημαίνει ότι ο αριθμός των 25 προσπαθειών για να τρέξει το transaction είναι μικρός, καθώς όταν ένα transaction κάνει abort λόγω conflict μπορεί αν προσπαθήσει ξανά, να εκτελεστεί επιτυχώς. Προκύπτει επομένως ότι αυτός ο αλγόριθμος μπορεί να επωφεληθεί με το να προσπαθεί περισσότερες φορές να εκτελέσει μια δοσοληψία πριν πάει στο global lock. Αυτό δεν ισχύει για τα capacity aborts, όπου όσες φορές και να προσπαθήσει μπορεί να αποτυγχάνει πάντα.

Παρατηρούμε επίσης ότι στο haci3 μηχάνημα, ο αριθμός των capacity aborts είναι μεγαλύτερος από αυτό στον broady. Αυτό ισχύει και για τους προηγούμενος αλγορίθμους, ωστόσο σε αυτόν τον αλγόριθμο είναι πιο φανερό. Ο κύριος λόγος για τον οποίο συμβαίνει αυτό είναι το μέγεθος της cache. Το broady σύστημα που αποτελείται από Intel Xeon E5-2699 v4 έχει 55MB cache για κάθε επεξεργαστή, ενώ το haci3 σύστημα που αποτελείται από Intel Xeon E5-2697 v3 έχει 35MB cache. Όπως ειπώθηκε στο κεφάλαιο για το transactional memory, οι read και write buffers κρατούνται στην cache.

Για να μειωθεί λοιπόν ο αριθμός των capacity aborts, δεδομένου ότι και εκεί υπάρχει χώρος για βελτίωση, πρέπει να μικρύνει το μέγεθος των transaction. Σε αυτόν τον αλγόριθμο ωστόσο υπάρχει το εξής πρόβλημα. Ας χωρίσουμε τον κώδικα του iteration σε δύο μέρη.

```
foreach (Point p : ws) {
      { //Part 1
            Point q = tree.find(p)
            Triangle t = findContainingTriangle(p, q);
            Cavity c = new Cavity(t, p);
            c.expand();
            c.retrianglate();
      }
      { //Part 2
            m.updateMesh(c);
            if (*)
            tree.update(...)
      }
}
```

Οι κλήσεις που γίνονται πάνω στην Cavity, είναι αναδρομικές. Αυτό σημαίνει ότι η εύρεση όλων των τριγώνων που επηρεάζονται από την εισαγωγή ενός κόμβου, μπορεί να χρειάζονται την αναδρομική κλήση κάποιων συναρτήσεων της κοιλότητας. Αν οι περιοχές του transaction είναι μέσα στον κώδικα των συναρτήσεων της κοιλότητας τότε σίγουρα θα υπάρχουν φωλιασμένα transactions. Όπως ειπώθηκε και στο κεφάλαιο για το transactional memory, το RTM υποστηρίζει μέχρι 7 φωλιασμένα transactions. Ωστόσο, από την στιγμή που δεν μπορούμε να είμαστε σίγουροι ότι αυτό ο αριθμός δεν ξεπεραστεί για κάποιο τρίγωνο, δεν μπορούμε να εισάγουμε transactional κώδικα μέσα στην αναδρομική κλήση καθώς αν αριθμός αυτός ξεπεραστεί θα γίνεται συνέχεια abort. Συνεπώς κάθε λειτουργία πάνω στο δέντρο και στην κοιλότητα θα αποτελεί ένα ξεχωριστό transaction.

Δεδομένου λοιπόν του παραπάνω fine grain σχήματος, προκύπτουν τα δύο μέρη κώδικα. Στο πρώτο μέρος όλες οι κλήσεις διαβάζουν από τα δεδομένα και αρχικοποιούν τις τοπικές δομές της κοιλότητας. Γίνεται επεξεργασία και εύρεση των ακμών που θα διαγραφούν και των νέων ακμών που θα δημιουργηθούν, και στο δεύτερο μέρος γράφονται όλες οι αλλαγές στα κοινά δεδομένα. Αν έχουμε δύο νήματα t1 και t2, που επεξεργάζονται δύο κόμβους n1 και n2, τότε κάθε νήμα θα εκτελέσει με την σειρά τις παραπάνω δοσοληψίες. Η τελική σειρά όμως που θα φανεί στα δεδομένα δεν είναι γνωστή. Ωστόσο, αν οι δύο κοιλότητες που επεξεργάζονται τα δύο νήματα έχουν εξαρτήσεις μεταξύ τους τότε αν το t1 τελειώσει το πρώτο μέρος, και πριν πάει στο δεύτερο μέρος τελειώσει και το t2 το πρώτο μέρος του, τότε και τα δύο νήματα έχουν διαβάσει και έχουν επεξεργαστεί δεδομένα, που κάποια μπορεί να είναι κοινά μεταξύ τους. Αυτό σημαίνει ότι όταν πάνε να εκτελέσουν το δεύτερο μέρος, τότε αυτός που θα εκτελέσει δεύτερος θα πάει να εφαρμόσει αλλαγές σε ‘’παλαιά’’ δεδομένα που έχουν αλλαχτεί από το άλλο νήμα. Η κοιλότητα του έχει αλλάξει και δεν είναι αυτή που διάβασε στο πρώτο μέρος. Για να προστατευθεί αυτή η εκτέλεση δημιουργείται μια validation φάση.

```
TRY_AGAIN:
	transaction_start();
	Point q = tree.find(p);
	transaction_end();

	transaction_start();
	Triangle t = findContainingTriangle(p, q);
	transaction_end();
	if (/*Node is not contained in the graph*/){
		return;
	}

	Cavity c(t, p);
	transaction_start();
	c.init(q, p);
	transaction_end();

	transaction_start();
	c.build();
	transaction_end();

	transaction_start();
	if (!c.validateCavity()){
		transaction_end();
		goto TRY_AGAIN;
	}
	c.update(); /* m.updateMesh(c)*/
	tree.update();
	transaction_end();
```

Κάθε λειτουργία πάνω στην κοιλότητα εκτελείται σε ένα transaction, εκτός από την φάση της ανανέωσης των κοινών δεδομένων που γίνεται σε μια δοσοληψία. Κατά την διάρκεια των κλήσεων πάνω στην κοιλότητα, διατηρείται ένας πίνακας με αναφορές σε κόμβους. Κάθε κόμβος που συμμετέχει στην δημιουργία της κοιλότητας εισάγεται στον πίνακα αυτό. Στην validate φάση, απλά διαβάζεται όλος ο πίνακας (μπαίνουν στο read set όλοι κόμβοι και προκαλούν πιθανά conflict abort) και αν κάποιος κόμβος έχει διαγραφεί, τότε το iteration πάει να εκτελεστεί από την αρχή για να δημιουργηθεί και πάλι η κοιλότητα με τα νέα δεδομένα. Αν ο προς εξέταση κόμβος έχει διαγραφεί τελείως, τότε δεν χρειάζεται να γίνει κάτι παραπάνω.

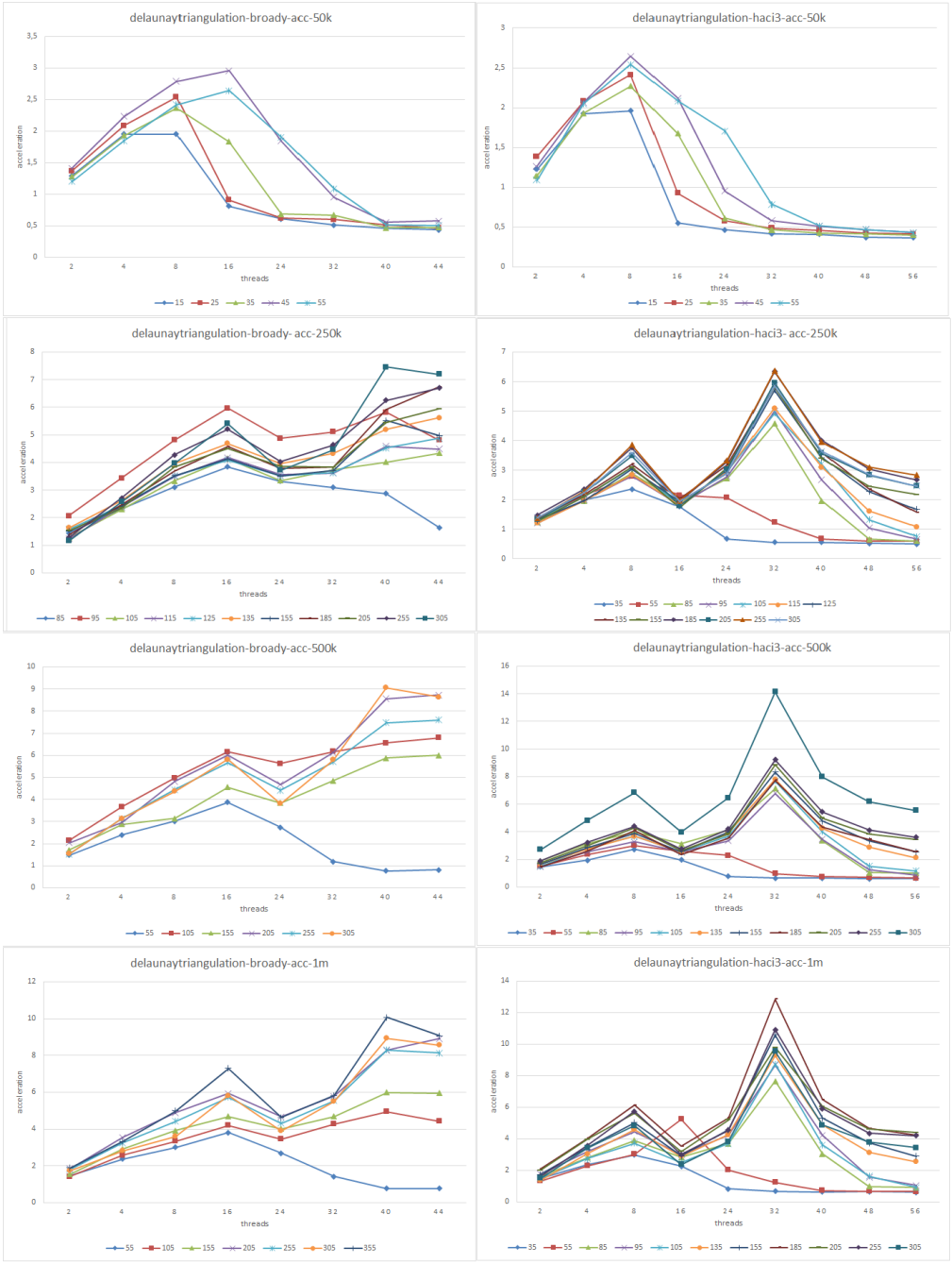

Figure 20: Επιτάχυνση της εκτέλεσης στον del. Triangulation με TSX σε fine grain σχήμα

Υλοποιώντας αυτήν την λογική έχουμε τις παραστάσεις στην Figure 20. Οι διαφορετικές γραφικές παραστάσεις αναφέρονται στον αριθμό που θα προσπαθήσει να εκτελεστεί ένα transaction. Παρατηρείται ένα ξεκάθαρο κέρδος όσο ο αριθμός αυτός

αυξάνει, καθώς τα περισσότερα aborts οφείλονταν σε conflicts. Οι γρηγορότεροι χρόνοι για το broady μηχάνημα έγιναν όλοι στα 44 νήματα (όλοι οι πυρήνες με ένα νήμα ανά πυρήνα, χωρίς HyperThreading) στις 185 με 205 προσπάθειες εκτός από το μικρό input των πενήντα χιλιάδων κόμβων, ενώ για το haci3 μηχάνημα έγιναν στα 32 νήματα (οι δύο πρώτοι πυρήνες HyperThreaded) στις 105~125 προσπάθειες. Το haci3 δεν φαίνεται να κερδίζει από τις παραπάνω προσπάθειες να εκτελέσει ένα transaction σε σχέση με το broady, καθώς όπως είδαμε και πριν, έχει μικρότερη cache που σημαίνει ότι θα και έχει σταθερά παραπάνω capacity aborts.

Στις εκτελέσεις στο broady μηχάνημα βλέπουμε μια σταθερή κάμψη στην απόδοση όταν τα νήματα αυξάνονται σε 24. Στα 24 νήματα υπάρχει Non Uniform Memory Access καθώς 22 νήματα εξυπηρετούνται από τους 22 πυρήνες ενός επεξεργαστή, και τα άλλα δύο νήματα εξυπηρετούνται από τον άλλον. Αυτό προκαλεί μια σταθερή κάμψη στην απόδοση σε όλες τις εκτελέσεις. Το ίδιο ακριβώς φαινόμενο παρατηρείται και στην haci3 εκτέλεση για 16 νήματα, όπου στο ένα chip εξυπηρετούνται τα 14, και στο άλλο τα 2. Φαίνεται λοιπόν ότι αλγόριθμος αυτός υποφέρει αρκετά παραπάνω από τους δύο προηγούμενους όταν υπάρχει NUMA.

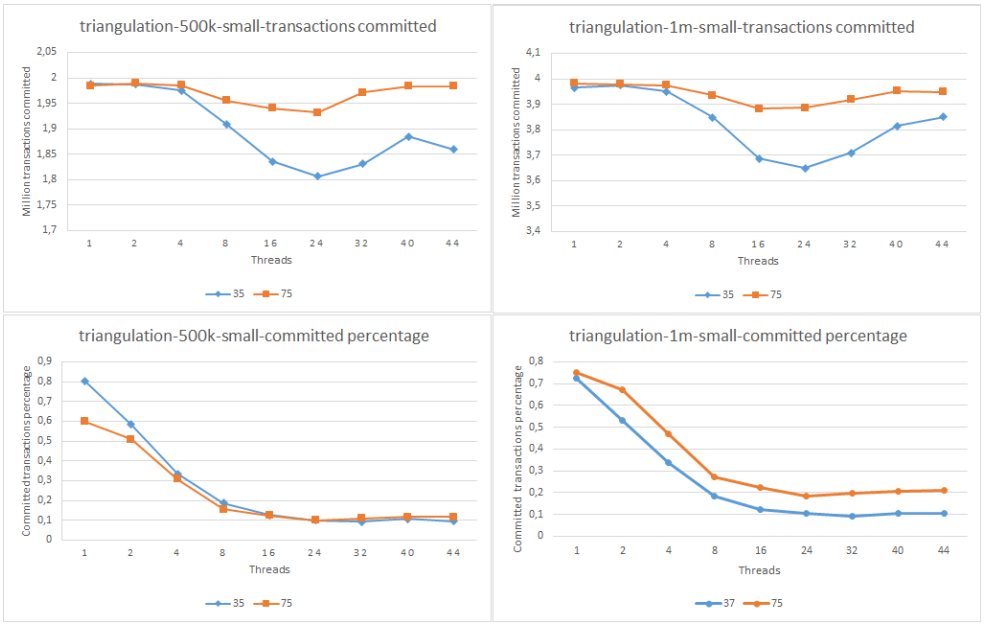

Figure 21: Ολοκληρωμένα transactions για del. Triangulation με TSX σε fine grain σχήμα στο broady 44 νημάτων

Στην Figure 21 βλέπουμε ότι μόνο μετά τα 4 νήματα οι περισσότερες προσπάθειες εκτέλεσης του transaction φαίνεται να χρησιμοποιούνται από το σύστημα. Αυτό είναι σε συμφωνία με την Figure 19, όπου τα conflict aborts αρχίζουν να αυξάνονται από τα 8 νήματα και πάνω. Το κέρδος στις περισσότερες προσπάθειες εκτέλεσης φαίνεται ιδιαίτερά στο μεγάλο input. Η εικόνα των capacity και conflict aborts δεν αλλάζει, με τα conflict aborts να είναι πολλές φορές περισσότερα από τα capacity, τα οποία μειώνονται λόγω και της μείωσης του μεγέθους των transaction. Τα conflict aborts τώρα είναι περισσότερα σε απόλυτο αριθμό έως και 3 με 4 φορές. Αυτό συμβαίνει διότι τα conflict aborts δεν είναι κάτι που μπορείς να το αποφύγεις καθώς είναι μέρος του αλγορίθμου, και όσο προσπαθείς να εκτελείς ένα transaction, αν υπάρχει εξάρτηση, αυτό δεν πρόκειται να κάνει ποτέ commit. Ωστόσο, το κέρδος του να προσπαθείς συνέχεια προκύπτει ότι από το ότι η εναλλακτική είναι να πάνε τα νήματα σε global lock. Αυτό μπορεί για μικρό αριθμό νημάτων να μην επηρεάζει την απόδοση, ωστόσο σε μεγάλο αριθμό νημάτων δεν επιτυγχάνεται η καλύτερη επιτάχυνση.

Στην Figure 22 έχουμε την εκτέλεση στο broady με HyperThreading. Παρατηρούμε ότι ο αλγόριθμος δεν κερδίζει κάτι, καθώς η αύξηση σε 88 νήματα προκαλεί μείωση στην επιτάχυνση και πιο αργή εκτέλεση. Ιδιαίτερα όταν ο οι φορές που θα προσπαθήσει ένα

transaction να εκτελεστεί είναι λίγες, η αύξηση των νημάτων έχει πολύ αρνητικές συνέπειες, καθώς παραπάνω νήματα σημαίνει περισσότερα conflict aborts, και όταν δεν προσπαθεί πολλές φορές να εκτελεστεί ένα transaction, τότε πολλές εργασίες πάνε στο global lock για να ολοκληρωθούνε. Αυτό φαίνεται και στα στατιστικά της Figure 23.

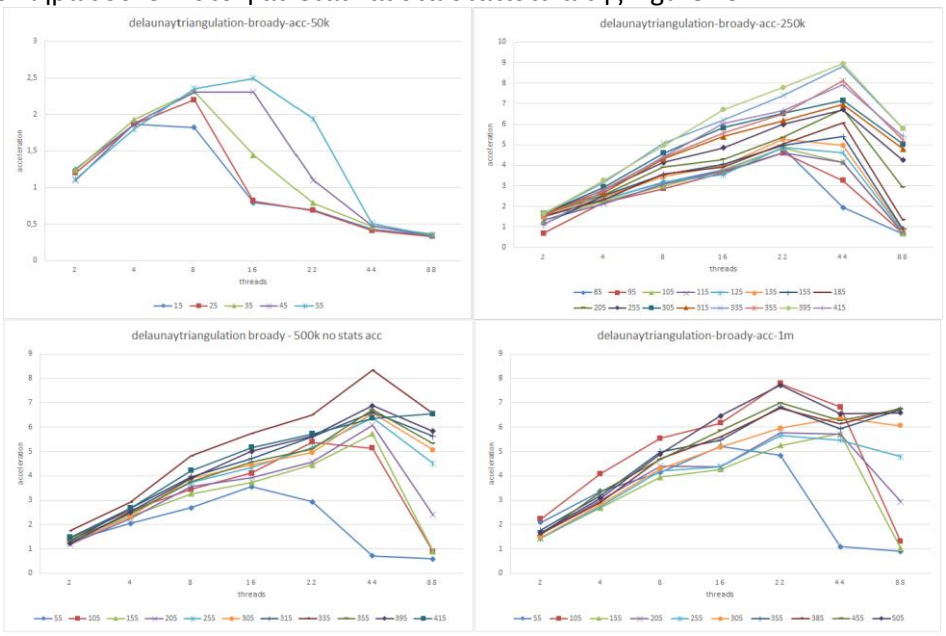

Figure 22: Επιτάχυνση της εκτέλεσης του del. Triangulation με TSX σε fine grain σχήμα στο broady 88 νημάτων

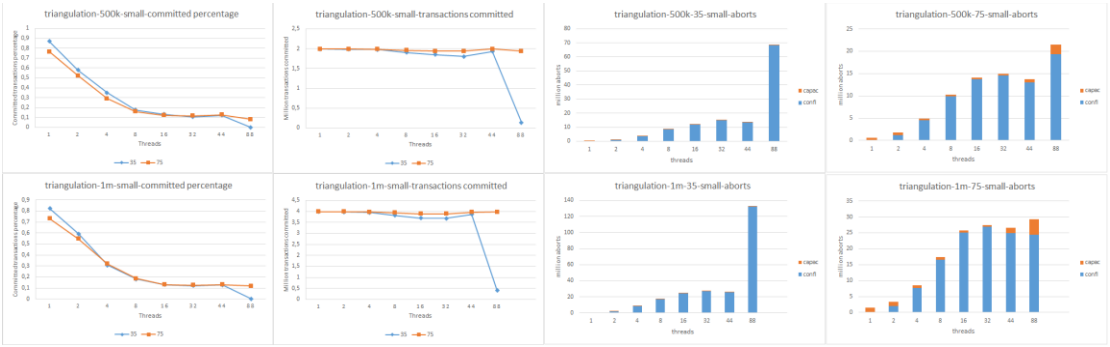

Figure 23: Committed και aborted transactions για del. Triangulation με TSX σε fine grain σχήμα στο broady 88 νημάτων

5.4 Delaunay Mesh refinement

Δεδομένου ενός τριγωνοποιημένου γράφου, ο αλγόριθμος Delaunay mesh refinement χρησιμοποιείται για να τον απαλλάξει από τρίγωνα που έχουν μια ανεπιθύμητη ιδιότητα. Η είσοδος του αλγορίθμου είναι συνήθως το αποτέλεσμα του Delaunay mesh triangulation, και η έξοδος του είναι ένας τριγωνοποιημένος γράφος, κάτω από τον κανόνα τριγωνοποίησης που αναφέρθηκε στην προηγούμενα ενότητα, όπου κανένα από τα τρίγωνα του γράφου δεν έχει την ανεπιθύμητη ιδιότητα.

Στην υλοποίηση του αλγορίθμου που θα εξεταστεί, όλα τα τρίγωνα του γράφου θα πρέπει να μην έχουν καμία γωνία κάτω από 30° μοίρες. Στην Figure 1 φαίνεται έναν τέτοιο παράδειγμα, όπου στην αριστερή εικόνα τα μπλε τρίγωνα είναι ανεπιθύμητα, και στην δεξιά εικόνα έχουν αντικατασταθεί από καινούρια χωρίς την ιδιότητα αυτή, και αποτελώντας ταυτόχρονα σωστή τριγωνοποίηση κατά Delaunay.

Όπως αναφέρθηκε στην ενότητα 2.2, ο αλγόριθμος δουλεύει στην λογική της δουλειάς εργασιών. Ο γράφος διαβάζεται, και όλα τα κακά τρίγωνα εισάγονται σε μια worklist. Σε κάθε

στοιχείο του αλγορίθμου θα εφαρμοστεί ένας refinement αλγόριθμος, όπου το τρίγωνο αυτό (οι συνδέσεις των κόμβων) θα διαγραφεί, και οι κόμβοι που το αποτελούν, μαζί με κάποιους γειτονικούς κόμβους, θα σχηματίσουν εκ νέου τρίγωνα. Η διαδικασία αυτή ονομάζεται retriangulation. Από τα καινούρια τρίγωνα που θα δημιουργηθούν, αν κάποιο από αυτά συνεχίζει να είναι κακό, τότε θα εισαχθεί εκ νέου στην λίστα.

```
Mesh m = /* read input mesh */;
Workset ws = new Workset(m.getBad());
foreach (Triangle t : ws) {
       Cavity c = new Cavity(t);
       c.expand();
       c.retriangulate();
       m.updateMesh(c);
       ws.add(c.getBad());
}
```

Η Cavity είναι μια κλάση που κρατάει την δουλειά και τα δεδομένα που αντιστοιχούν σε ένα τρίγωνο. Η συνάρτηση `expand()` πάνω στην κοιλότητα του τριγώνου, είναι αυτή που βρίσκει όλα τα γειτονικά τρίγωνα που πρέπει να συμμετάσχουν στον ανασχηματισμό. Η συνάρτηση `retriangulate()` υλοποιεί τον αλγόριθμο της τριγωνοποίησης εισάγοντας έναν κεντρικό κόμβο μέσα στην κοιλότητα. Στην συνέχεια, οι αλλαγές γράφονται πάνω στον γράφο, και τα κακά τρίγωνα της κοιλότητας (αν υπάρχουν) εισάγονται εκ νέου στην λίστα εργασιών.

5.4.1 Galois

Στην υλοποίηση του αλγορίθμου από το Galois σύστημα, δεν χρησιμοποιούνται ειδικές δομές όπως στους προηγούμενος αλγορίθμους, Η επανάληψη πάνω στα κακά τρίγωνα γίνεται με έναν `unordered set iterator`, καθώς η σειρά με την οποία θα επεξεργαστούν τα τρίγωνα δεν έχει σημασία. Η διαφορετική σειρά μπορεί να έχει στο τέλος διαφορετικό γράφο, ωστόσο σημασία έχει να αποτελεί σωστή refined τριγωνοποίηση. Τα δεδομένα αναπαρίστανται σε έναν γράφο, όπου η ερμηνεία είναι αντίστροφη με αυτήν των εικόνων. Οι κόμβοι του γράφου είναι τα τρίγωνα, και οι συνδέσεις μεταξύ κόμβων είναι τα γειτονικά τρίγωνα στον γράφο.

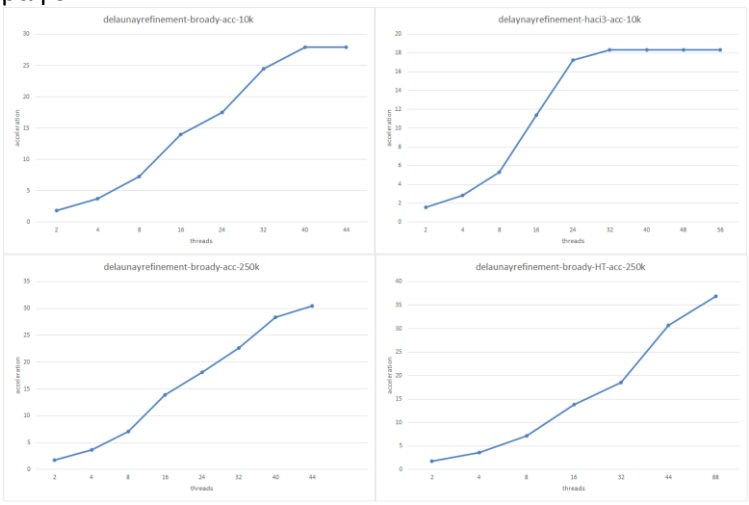

Figure 24: Επιτάχυνση της εκτέλεσης του galois Delaunay refinement στην αύξηση των νημάτων

Η κοιλότητα που αρχικοποιείται για κάθε τρίγωνο, αποτελεί τα τοπικά δεδομένα του τριγώνου προς εξέταση. Η αρχικοποίηση γίνεται διαβάζοντας ένα κομμάτι του τριγώνου από τον γράφο που βλέπουν όλα τα νήματα, και στην συνέχεια οι αλλαγές γράφονται πάνω στον γράφο. Δυο τρίγωνα λοιπόν, μπορούν να εξεταστούν ταυτόχρονα μόνο αν οι δύο κοιλότητες δεν έχουν κοινά τρίγωνα. Ωστόσο, ο διαθέσιμος παραλληλισμός φθίνει όσο περισσότερα

τρίγωνα μετασχηματίζονται και αυτό γιατί σε πολλές περιπτώσεις, η μετατροπή παράγει νέα κακά τρίγωνα τα οποία αναγκάστηκα επεξεργάζονται σειριακά.

Στην Figure 24 βλέπουμε ότι ο αλγόριθμος παρουσιάζει εξαιρετική παράλληλη συμπεριφορά, πολύ καλύτερη από αυτήν του triangulation. Στον refinement αλγόριθμο το σώμα ενός iteration είναι πολύ πιο ελαφρύ από αυτό του triangulation, καθώς δεν χρησιμοποιούνται πρόσθετες δομές για επιτάχυνση της εκτέλεσης. Αυτό φαίνεται και από την συμπεριφορά της εκτέλεσης όταν χρησιμοποιείται το HyperThreading, όπου και στο haci3 και στο broady μηχάνημα, ο αλγόριθμος κερδίζει σημαντικά στην αύξηση των νημάτων, κάτι το οποίο δεν ίσχυε για κανένα από τους προηγούμενους αλγορίθμους. Υπάρχει μια ελάχιστη κάμψη στην επιτάχυνση όταν τα νήματα αυξάνονται για να χρησιμοποιηθεί και το δεύτερο chip του αντίστοιχου συστήματος, ωστόσο είναι αρκετά μακριά από τα αντίστοιχα αποτελέσματα στον Delaunay triangulation.

5.4.2 TSX

Όπως και στους προηγούμενους αλγορίθμους, ορίζουμε όλο τον κώδικα του iteration ως ένα transaction.

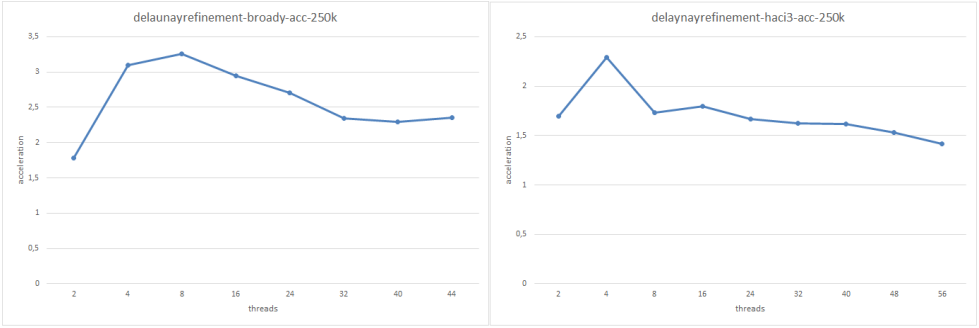

Figure 25: Επιτάχυνση της εκτέλεσης του Del. Refinement αλγορίθμου όταν χρησιμοποιείται TSX σε coarse grain σχήμα.

Όπως είναι αναμενόμενο, η απόδοση δεν φτάνει στα επιθυμητά επίπεδα. Και σε αυτήν την περίπτωση, όπως και στην προηγούμενη, ο κύριος λόγος δεν είναι τα capacity aborts, αλλά τα conflict aborts. Στην Figure 25, οι εκτελέσεις γίνονται με 25 προσπάθειες να εκτελεστεί το transaction πριν πάει στο global lock.

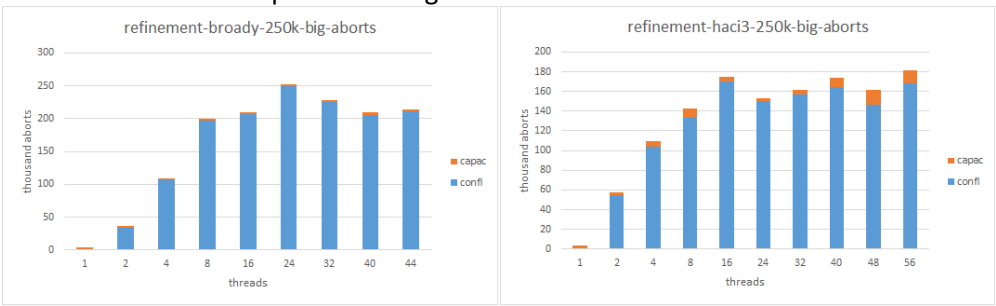

Figure 26: TSX committed και aborted transactions στον Del. refinement με TSX σε coarse grain σχήμα

Στην Figure 26, φαίνεται ότι το πρόβλημα είναι στον μεγάλο αριθμό των conflict aborts. Τα ολοκληρωμένα transactions εδώ είναι πρακτικά μηδενικά. Για να αντιμετωπιστεί αυτό το πρόβλημα, θα αυξήσουμε τον αριθμό των προσπαθειών ενός TSX transaction, και θα σπάσουμε τον κώδικα του iteration σε μικρότερα transactions. Με αυτόν τον τρόπο, δεν μειώνουμε μόνο τα capacity aborts, αλλά βελτιστοποιούμε και την εκτέλεση, καθώς όταν ένα μεγάλο transaction κάνει abort, πρέπει να κάνει όλη την δουλειά από την αρχή. Τέλος, και σε αυτήν την εκτέλεση, ο αριθμός των capacity aborts, είναι σταθερά μεγαλύτερος στο haci3 μηχάνημα.

Όπως και στην περίπτωση του Delaunay triangulation, ο κώδικας του iteration χωρίζεται σε δύο μέρη. Στο πρώτο μέρος οι συναρτήσεις διαβάζουν από τα κοινά δεδομένα, και στο δεύτερο μέρος οι συναρτήσεις γράφουν σε αυτά.

```
TRY_AGAIN:
      if (/*Node is not contained in the graph*/) return;
      Cavity cav();

      transaction_start(&fallback_lock,number_of_aborts);
      cav.initialize(triangle);
      transaction_end(&fallback_lock);

      transaction_start(&fallback_lock,number_of_aborts);
      cav.build();
      transaction_end(&fallback_lock);

      cav.computePost();

      transaction_start(&fallback_lock,number_of_aborts);
      if (!cav.validateCavity()){
            transaction_end(&fallback_lock);
            goto TRY_AGAIN;
      }
      cav.update(triangle);
      transaction_end(&fallback_lock);
```

Η αρχικοποίηση της κλάσης της κοιλότητας, και οι συναρτήσεις *initialize(triangle)*, *build()* και *computePost()* μόνο διαβάζουν από τα κοινά δεδομένα. Στην πραγματικότητα η computePost(), διαβάζει και επεξεργάζεται τα δεδομένα στην τοπική κλάση Cavity, τα οποία έχουν φέρει οι δύο προηγούμενες κλήσεις. Στην αρχή δηλαδή, όλα τα απαραίτητα δεδομένα αρχικοποιούνται στην τοπική κλάση Cavity, και η επεξεργασία γίνεται πάνω σε αυτά, σε αντίθεση με τον Del. Triangulation αλγόριθμο όπου η επεξεργασία απαιτούσε συγχρονισμό. Αυτή η μικρή διαφορά, κάνει το σώμα του αλγορίθμου αυτού αρκετά μικρότερο, προκαλώντας και μικρότερο αριθμό από capacity aborts.

Στον αλγόριθμο αυτό αντιμετωπίζουμε το ίδιο πρόβλημα που είχαμε και στο triangulation, όσον αφορά το isolation των transaction. Μπορεί η επεξεργασία των κοινών δεδομένων να γίνεται τοπικά ωστόσο, όταν ένα νήμα t1 πάει να καλέσει την *update(triagnle)*, που γράφει τις αλλαγές στα κοινά δεδομένα, μπορεί να έχει προλάβει να κάνει update ένα άλλο νήμα t2 που επεξεργαζόταν μια κοιλότητα με εξαρτήσεις με την κοιλότητα του t1. Αυτό προκαλεί την κοιλότητα του t1 να μην ισχύει πλέον. Σε αυτήν την περίπτωση, το νήμα t1 πρέπει να εκκινήσει από την αρχή, και αν το τρίγωνο του δεν υπάρχει πια, να επιστρέψει αμέσως.

Το παραπάνω σχήμα υλοποιείται με τον ίδιο τρόπο που γίνεται και στο triangulation. Καθ' όλη την διάρκεια των συναρτήσεων πάνω στην κοιλότητα, κάθε κόμβος που εισάγεται σε αυτήν εισάγεται και σε ένα ακόμη διάνυσμα. Πριν το νήμα πάει να γράψει τις αλλαγές, καλεί την *validateCavity()* που ελέγχει αν μπορεί να προχωρήσει. Ο έλεγχος αυτός αποτελείται από το διάβασμα όλων των κόμβων (εισαγωγή στο read set και πρόκληση πιθανών conflict), και τον έλεγχο για το αν υπάρχουν ακόμα οι κόμβοι στον γράφο. Αν δεν υπάρχουν, τότε το νήμα πρέπει να ξαναδιαβάσει την γειτονιά του τριγώνου, και να ξανακάνει επεξεργασία. Αν όλοι οι κόμβοι είναι ζωντανοί, τότε μπορεί να γράψει τις αλλαγές.

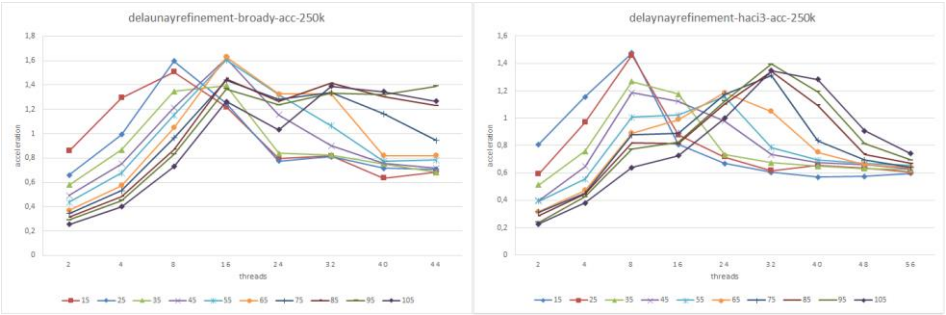

Figure 27: Επιτάχυνση της εκτέλεσης του Del. Refinement με TSX σε fine grain σχήμα

Στην Figure 27 βλέπουμε τα αποτελέσματα αυτής της υλοποίησης. Κάθε γραφική αναφέρεται στον αριθμό των προσπαθειών των transactions να κάνουν commit. Αυτό που φαίνεται είναι ότι δεν υπάρχει καλή κλιμάκωση στην αύξηση των νημάτων. Οι εκτελέσεις κερδίζουν στην αύξηση των προσπαθειών, αλλά η επιτάχυνση δεν φτάνει στα επίπεδα των προηγούμενων αλγορίθμων. Στην πραγματικότητα, η επιτάχυνση είναι χειρότερη από όταν χρησιμοποιείται TSX σε coarse grain σχήμα.

Η εκτέλεση στην Figure 27, γίνεται χωρίς συλλογή στατιστικών. Στο σχήμα που περιγράφεται στη ενότητα 3.5.5 κάθε νήμα κρατάει τα στοιχεία της εκτέλεσης των transaction σε ανεξάρτητες μεταβλητές με τα άλλα νήματα, και η συλλογή γίνεται στο τέλος. Όλες οι μέχρι τώρα εκτελέσεις έγιναν με εκείνο το σχήμα, το οποίο επιτρέπει να συλλέγονται τα στατιστικά μιας εκτέλεσης με ελάχιστο overhead. Ωστόσο η πρώτη έκδοση της κλάσης συλλογής στατιστικών χρησιμοποιούσε global μεταβλητές, όπου τα νήματα έκαναν αυξήσεις με ατομικές εντολές. Αυτό το σχήμα εισάγει κάποιο overhead στην εκτέλεση, το οποίο είναι αρκετά σημαντικό, και γι' αυτό αλλάχθηκε. Ωστόσο, τρέχοντας το παραπάνω σχήμα συγχρονισμού, μαζί με συλλογή στατιστικών με ατομικές εντολές υπάρχουν κάποια ενδιαφέροντα αποτελέσματα.

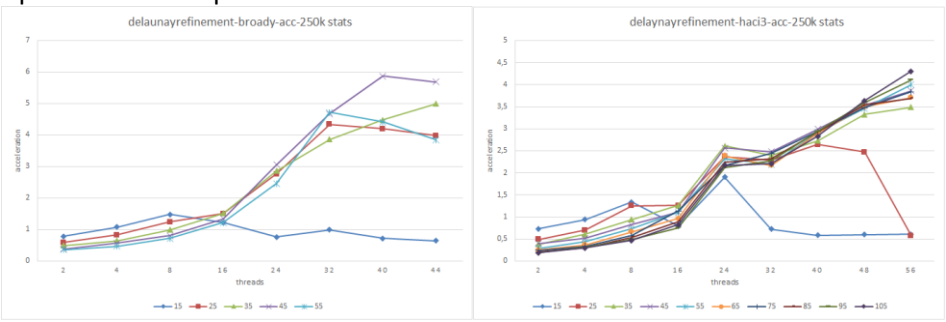

Figure 28: Επιτάχυνση της εκτέλεσης του Del. Refinement με TSX σε fine grain σχήμα, όταν συλλέγονται στατιστικά με ατομικές εντολές

Στην Figure 28, έχουμε την εκτέλεση του ίδιου αλγορίθμου με ατομικές εντολές για συλλογή στατιστικών. Παρατηρείται μια αρκετά μεγάλη διαφορά στην επιτάχυνση του αλγορίθμου σε σχέση με την προηγούμενη εκτέλεση. Θεωρητικά, η εκτέλεση με συλλογή στατιστικών θα πρέπει να μειώνει ακόμα περισσότερο την απόδοση του αλγορίθμου στην αύξηση των νημάτων, καθώς όλο και περισσότερα νήματα σειριοποιούνται πίσω από μια ατομική εντολή μνήμης σε κάθε επιτυχημένο ή αποτυχημένο transaction.

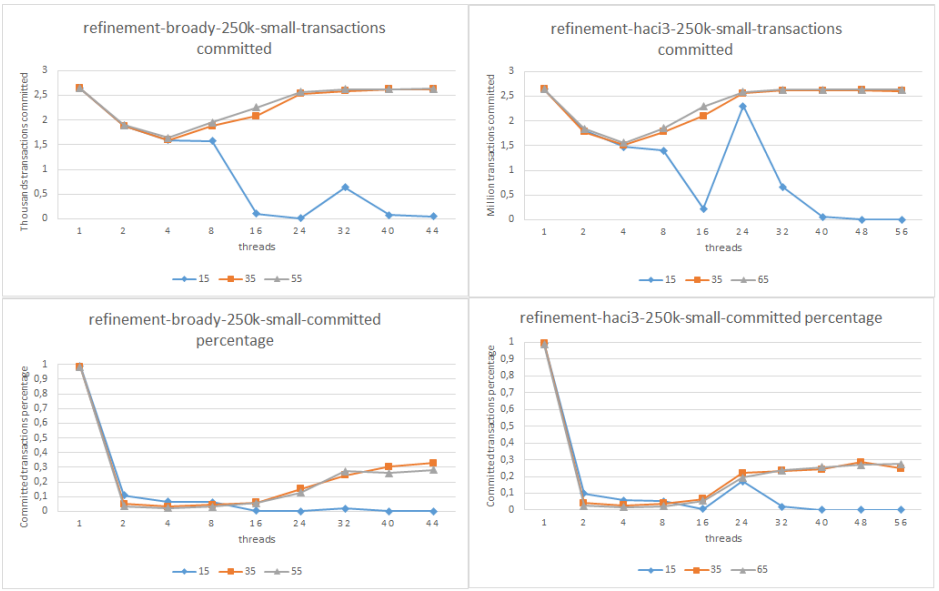

Figure 29: Ολοκληρωμένα transactions του Del. Refinement με TSX σε fine grain σχήμα όταν συλλέγονται στατιστικά με ατομικές εντολές

Στην Figure 29, φαίνονται τα επιτυχώς ολοκληρωμένα transactions της εκτέλεσης της Figure 28. Φαίνεται κατ' αρχήν ότι ο αλγόριθμος κερδίζει από την αύξηση των προσπαθειών ενός transaction. Τα aborts είναι σχεδόν όλα conflict aborts και τα capacity aborts είναι ασήμαντα.

Για να δούμε την διαφορά μεταξύ της εκτέλεσης μαζεύοντας στατιστικά με ατομικές και εντολές, σε σχέση με την εκτέλεση όταν δεν μαζεύονται στατιστικά, συγκρίνουμε τα επιτυχώς ολοκληρωμένα transactions και τα conflict aborts με μια εκτέλεση όπου μαζεύονται στατιστικά ανά νήμα σε σχέση με τις ατομικές εντολές. Μαζεύοντας στατιστικά ανά νήμα εισάγεται ελάχιστο overhead στην εκτέλεση, το οποίο θα προσομοιάζει την εκτέλεση που δεν μαζεύονται στατιστικά.

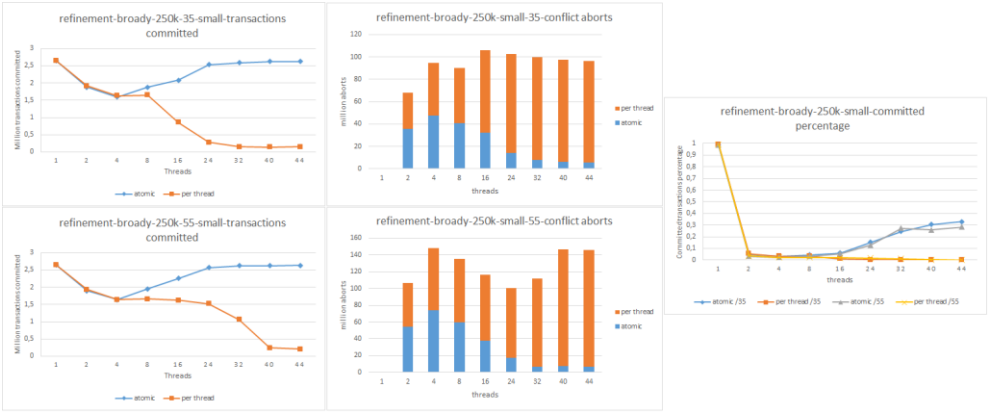

Figure 30: Ολοκληρωμένα transactions και conflict aborts του Del. Refinement για ατομικές εντολές και ανά νήμα συλλογή στατιστικών

Στην Figure 30, φαίνονται οι διαφορετικές εκτελέσεις, για 35 και 55 προσπάθειες αντίστοιχα. Οι `atomic` και `per thread` γραφικές αναφέρονται στον ίδιο ακριβώς αλγόριθμο με την μόνη διαφορά την κλάση συλλογής στατιστικών. Φαίνεται μια ξεκάθαρη διαφορά στα ολοκληρωμένα transactions μεταξύ των δύο εκτελέσεων και μια ακόμα μεγαλύτερη διαφορά φαίνεται και στα conflict aborts. Τα capacity aborts δεν επηρεάζονται καθώς η συλλογή στατιστικών γίνεται εκτός του κώδικα της δοσοληψίας.

Αυτό το οποίο συμβαίνει σε αυτό τον αλγόριθμο είναι αυτό το οποίο αναφέρεται και στο Optimization Manual από την Intel [35]. Όταν ένα transaction έχει αποτύχει την εκτέλεση του, και έχει πάει στο fallback μονοπάτι, ο έλεγχος δίνεται στον προγραμματιστή μαζί με

πληροφορίες για τον λόγο του abort. Όταν ο λόγος του abort είναι σύγκρουση δεδομένων (conflict abort), τότε προτείνεται η εκτέλεση να περιμένει πριν πάει να προσπαθήσει ξανά να τρέξει το transaction, και αυτό γιατί το άλλο transaction με το οποίο υπάρχει εξάρτηση και προκάλεσε το conflict abort, είναι πολύ πιθανό τρέχει ακόμη. Αυτό θα προκαλέσει conflict abort και πάλι το οποίο θα οδηγήσει σε μια συνεχή εξάρτηση μεταξύ των transaction.

Ο λόγος για τον οποίο στην έκδοση με ατομικές εντολές αυτό το φαινόμενο δεν παρατηρείται, είναι διότι η ατομική αύξηση μιας μεταβλητής προκαλεί μια σειριοποίηση πίσω από την εντολή μνήμης, η οποία εισάγει μια καθυστέρηση πριν την επανεκτέλεση του transaction.

Για να εξαλείψουμε αυτό το πρόβλημα, αλλάζουμε τον κώδικα του fallback μονοπατιού που φαίνεται στην ενότητα 3.5.5, έτσι ώστε να εισάγει μια καθυστέρηση.

```
static inline int transaction_start(pthread_spinlock_t * fallback_lock, int
number_of_tries){

     #ifdef RTM_GATHER_STATS
              ReportStat& stats = ReportStat::getInstance();
     #endif

             int status;
             int aborts=0;
             while(1){
                     while(*fallback_lock == 0);

                     __try_transaction(status){
                             if (*fallback_lock == 0)
__transaction_abort(0xff);
                             return number_of_tries - aborts;
                     }
                     __transaction_abort(0xff);
     #ifdef RTM_GATHER_STATS
                     stats.report_status(status);
     #endif
                     if (status & _XABORT_CONFLICT) usleep(10);

                     if (aborts++ == number_of_tries){
                             pthread_spin_lock(fallback_lock);
                             return number_of_tries - aborts;
                     }
             }
             return -1;
     }
```

Η μόνη αλλαγή στον κώδικα είναι η προσθήκη της *if-then clause*, που ελέγχει για τον αν ο λόγος του abort είναι conflict, και αν είναι τότε καθυστερεί την εκτέλεση του νήματος 10 μs. Τα 10μs προέκυψαν πειραματικά τρέχοντας διαφορετικές εκδόσεις του αλγορίθμου, και λαμβάνοντας υπόψη ότι όσο μεγαλύτερη είναι η καθυστέρηση, τόσο πιο αργή θα είναι και η εκτέλεση. Στην Figure 31, φαίνεται η σύγκριση των εκτελέσεων αυτών για 35, 75 και 185 προσπάθειες εκτέλεσης.

Figure 31: Ολοκληρωμένα transactions και conflict aborts του Del. Refinement με TSX σε fine grain σχήμα στην μεταβολή της καθυστέρησης επανεκτέλεσης των transaction

Figure 32: Επιτάχυνση της εκτέλεσης του Del. Refinement με TSX σε fine grain σχήμα, με καθυστέρηση 50μsec για επανεκτέλεση

Στην Figure 32 έχουμε τα αποτελέσματα της εκτέλεσης όταν εισάγεται καθυστέρηση στην επανεκτέλεση των transaction 50μsec, όπου και φαίνεται μια καλύτερη απόδοση σε σχέση με την Figure 28. Στις δεξιά παραστάσεις φαίνονται τα conflict aborts του αλγορίθμου με καθυστέρηση πριν την επανεκτέλεση, σε σχέση με την συλλογή στατιστικών με ατομικές εντολές, και τα ποσοστά των committed transactions.

Ιδιαίτερα, ο αλγόριθμος με ατομικές εντολές υποφέρει στα λίγα νήματα καθώς η σειριοποίηση πίσω από την ατομική αύξηση είναι μικρότερη, οδηγώντας και σε μικρότερη καθυστέρηση.

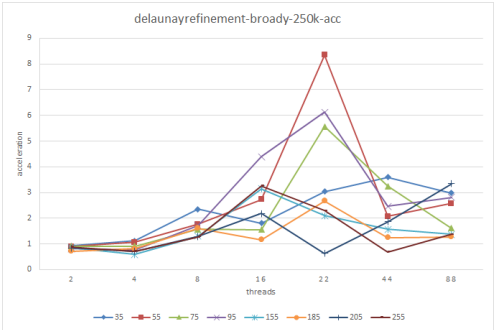

Figure 33: Επιτάχυνση της εκτέλεσης του Del.Refinement με TSX σε fine grain σχήμα, και καθυστέρηση 10μsec για επανεκτέλεση

Στην Figure 33 φαίνεται η εκτέλεση του ίδιου αλγορίθμου με 10μsec καθυστέρηση πριν την επανεκτέλεση των transaction. Η μεγάλη διαφορά μεταξύ των δύο είναι η κλιμάκωση στα

μεγαλύτερα νήματα. Στα 50μsec, υπάρχουν αρκετά καλύτεροι χρόνοι, καθώς τα περισσότερα νήματα έχουν και παραπάνω εξαρτήσεις μεταξύ τους, και χρειάζεται να περιμένουν περισσότερο. Ωστόσο, οι καλύτεροι χρόνοι δεν επιτυγχάνονται σε μεγάλο αριθμό νημάτων, αλλά μόλις στα 22 νήματα που είναι ένα chip χωρίς HyperThreading.

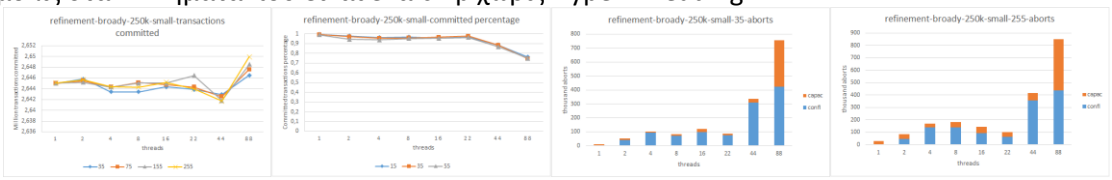

Figure 34: TSX committed και aborted transactions του Del. Refinement με TSX σε fine grain σχήμα, και καθυστέρηση 10μsec

Οι καλύτεροι χρόνοι για αυτόν τον αλγόριθμο δεν έχουν συγκεκριμένο προφίλ όπως στον Delaunay triangulation. Ορισμένες εκτελέσεις το επιτυγχάνουν στα 22 νήματα, και ορισμένες στα 44. Στην Figure 34, φαίνεται επίσης και η μεγάλη αύξηση των aborts όταν εξυπηρετούνται 2 νήματα ανά πυρήνα. Επίσης, ο αλγόριθμος αυτός υποφέρει και όταν υπάρχει NUMA.

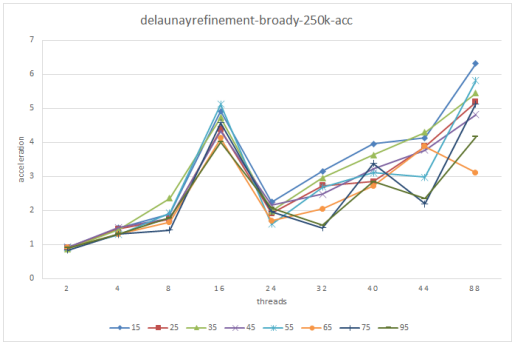

Figure 35: Απόδοση του Del. Refinement σε NUMA

Στην Figure 35, βλέπουμε πόσο πέφτει η απόδοση του αλγορίθμου όταν τα νήματα είναι 24 (ένας επεξεργαστής γεμάτος, και μόλις 2 νήματα στον άλλο) όταν η καλύτερη επιτάχυνση επιτυγχάνεται στα 22 νήματα σύμφωνα με την Figure 32.

Τέλος, η επιτάχυνση που φτάνει ο αλγόριθμος όταν χρησιμοποιείται TSX είναι αρκετά μακριά από αυτή του Galois. Αυτό που φαίνεται συγκεκριμένα είναι η εξαιρετική κλιμάκωση που έχει το semantic commutativity (μια software λύση για την επίλυση του συγχρονισμού), όταν τα νήματα αυξάνονται πέρα από τον έναν πυρήνα. Το TSX φαίνεται να υποφέρει αρκετά σε αυτόν τον αλγόριθμο από conflict aborts, και ο έλεγχος μέσω της cache υποφέρει ακόμα περισσότερο όταν υπάρχει NUMA. Ωστόσο, η αύξηση πέρα από τα 44 νήματα μέσω της HyperThreading τεχνολογίας αυξάνει την απόδοση καθώς το σώμα του iteration εδώ είναι αρκετά μικρό. Αυτό δεν ίσχυε στον triangulation αλγόριθμο, ο οποίος είχε μεγαλύτερο πρόβλημα με capacity aborts.

Δεδομένου ότι και ο Delaunay triangulation αλγόριθμος υπέφερε από μεγάλο αριθμό conflict aborts, η καθυστέρηση πριν την επανεκτέλεση των transaction είναι αρκετά πιθανό να πετυχαίνει καλύτερη επιτάχυνση. Γι’ αυτό το λόγο τρέχουμε τον triangulation αλγόριθμο εισάγοντας καθυστέρηση μετά από conflict abort.

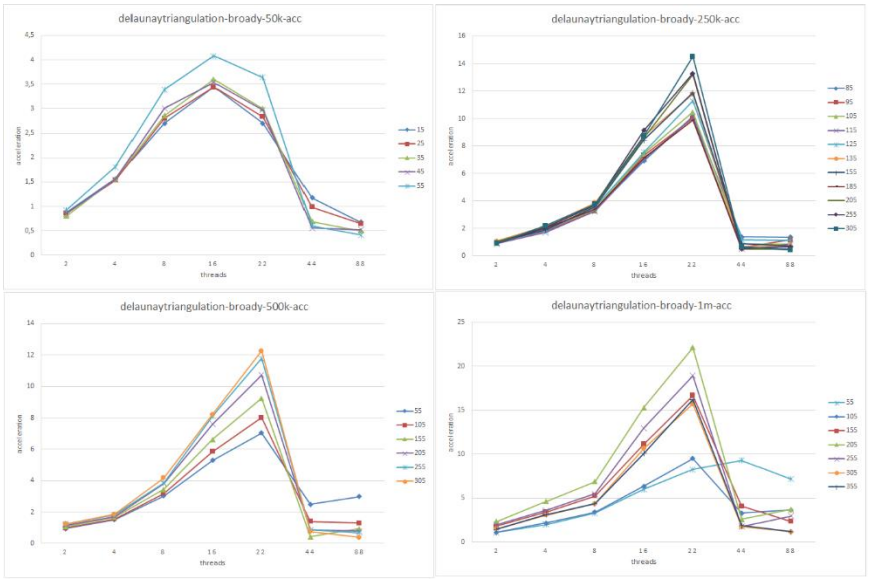

Figure 36:Επιτάχυνση της εκτέλεσης του Del. Triangulation με TSX σε fine grain σχήμα και 10μs καθυστέρηση στην επανεκτέλεση των transaction

Στην Figure 36, φαίνονται αρκετά καλύτερα αποτελέσματα σε σχέση με την Figure 22. Αυτά όμως δεν αντικατοπτρίζουν την πραγματικότητα και αυτό γιατί η φαινομενικά μεγάλη επιτάχυνση, προκύπτει λόγο του εξαιρετικά μεγάλου χρόνου εκτέλεσης για ένα νήμα. Ωστόσο, ο αλγόριθμος έχει κέρδος από την εισαγωγή της καθυστέρησής, καθώς οι καλύτεροι χρόνοι εκτέλεσης σε απόλυτο αριθμό σε αυτήν την έκδοση, είναι καλύτεροι από αυτούς χωρίς καθυστέρηση, για όλα τα input. Αυτός ο χρόνος ωστόσο, επιτυγχάνεται στα 22 νήματα για 55 προσπάθειες. Χωρίς την καθυστέρηση ο καλύτερος χρόνος επιτυγχάνεται σε μεγαλύτερο αριθμό προσπαθειών. Όπως συζητήθηκε και στο αντίστοιχο κεφάλαιο, ο αλγόριθμος αυτός υποφέρει αρκετά από NUMA και HyperThreading (Figure 23), και η εισαγωγή μιας sleep εντολής όταν τα aborts γίνονται πολλά, κατεβάζει ακόμα περισσότερο την απόδοση στα πολλά νήματα. Γι' αυτό το λόγο, η αντίστοιχη εκτέλεση με 50μsec καθυστέρηση έχει ακόμα χειρότερα αποτελέσματα από την Figure 36.

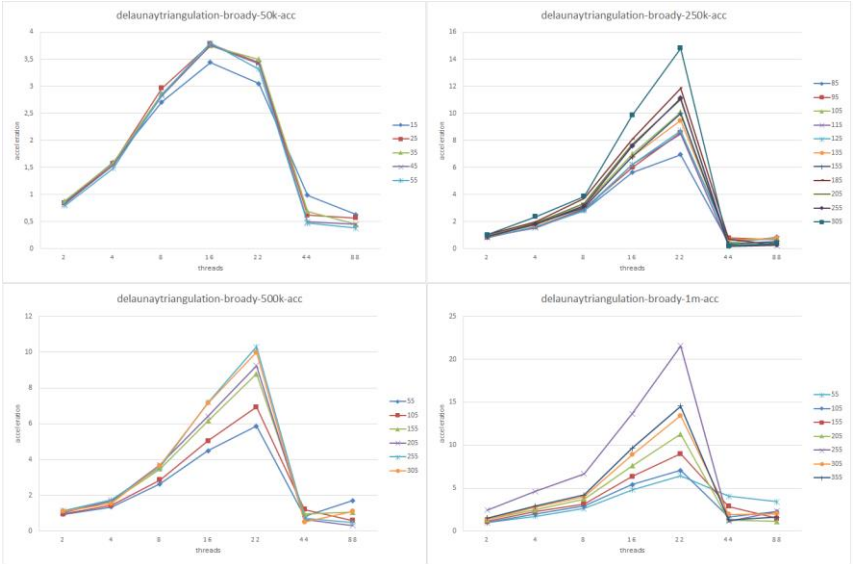

Figure 37:Επιτάχυνση της εκτέλεσης του Del. Triangulation με TSX σε fine grain σχήμα και 50μs καθυστέρηση στην επανεκτέλεση των transaction

Στην Figure 37 φαίνεται η εκτέλεση αυτή. Και πάλι οι μεγάλοι αριθμοί επιτάχυνσης οφείλονται στον εξαιρετικά μεγάλο χρόνο για ένα νήμα. Με 10μs καθυστέρηση, επιτυγχάνονται μικρότεροι καλύτεροι χρόνοι σε σχέση με αυτή στα 50μs. Ωστόσο, όταν η

καθυστέρηση είναι μεγαλύτερη, ο αλγόριθμος κλιμακώνει καλύτερα όσο αυξάνουν τα νήματα. Συγκεκριμένα, από τα 22 νήματα και πάνω, στα 50μs καθυστέρηση οι χρόνοι είναι καλύτεροι σε σχέση με τα 10μs. Το γεγονός όμως ότι η καθυστέρηση αυτή εφαρμόζεται άσχετα με τον αριθμό των νημάτων που χρησιμοποιούνται, καθιστά την εκτέλεση με τα 50μs αργή, και δεν επιτυγχάνει τους καλύτερους χρόνους σε απόλυτο αριθμό.

6 Συμπεράσματα

Η δυσκολία υλοποίησης του σχήματος συγχρονισμού από αλγόριθμο σε αλγόριθμο διαφέρει. Είδαμε πόσο εύκολο ήταν να γίνει στον barneshut και στον clustering αλγόριθμο, με τον πρώτο να παρουσιάζει εξαιρετική παράλληλη συμπεριφορά, και τον δεύτερο μόλις να ξεπερνάει τις 2 φορές γρηγορότερη εκτέλεση. Αντιθέτως, η υλοποίηση του συγχρονισμού στους Delaunay triangulation και refinement ήταν αρκετά διαφορετική, καθώς έπρεπε να εισαχθεί validation φάση και επαναφορά της επεξεργασίας του στοιχείου της worklist από την αρχή.

Ο τρόπος με τον οποίο αποφασίζει το Galois να λύσει το πρόβλημα του συγχρονισμού είναι να πάρει όλη την ευθύνη από τον προγραμματιστή, και λύσει το πρόβλημα κεντρικά, προσφέροντας του έτοιμες βιβλιοθήκες με δομές δεδομένων να χρησιμοποιήσει για τα προγράμματα του. Ο προγραμματιστικός κόπος αυτού του σχήματος είναι πολλές φορές μεγαλύτερος, αλλά δεν γίνεται από τον σχεδιαστή του αλγορίθμου αλλά από το Galois.

Το semantic commutativity που χρησιμοποιεί το Galois, εκμεταλλεύεται την σημασιολογία των κλήσεων πάνω στα κοινά δεδομένα και είναι πιο αποδοτικό από το Transactional Memory που κοιτάει τυφλές διευθύνσεις. Ο κώδικας μιας συνάρτησης δεν είναι τίποτα άλλο παρά μια σειρά από `read` και `write` εντολές σε διευθύνσεις. Αν η σειρά αυτών των εντολών μπορεί να αντιμετατεθεί σημασιολογικά, τότε και το transactional memory θα βρει ότι δεν υπάρχει σύγκρουση δεδομένων γιατί θα αναφέρονται σε διαφορετικές διευθύνσεις. Το semantic commutativity ωστόσο, επειδή μπορεί και ξέρει την σημασιολογία των κλήσεων βγάζει το συμπέρασμα για όλη την ομάδα των `read` και `write` εντολών της συνάρτησης, ενώ το transactional memory πρέπει αναγκάστηκα να κοιτάξει όλες τις διευθύνσεις μνήμης. Με αυτό τον τρόπο το semantic commutativity μπορεί και γλυτώνει χρόνο εκτέλεσης.

Ωστόσο, επειδή το semantic commutativity έρχεται μαζί με τις διαθέσιμες δομές, οι έλεγχοι γίνονται πάντα, χωρίς να μπορεί να ξέρει το Galois αν χρειάζονται, βάζοντας έτσι ένα βάρος στο Runtime σύστημα. Αντιθέτως, όταν ο συγχρονισμός γίνεται από τον προγραμματιστή του αλγορίθμου ο συγχρονισμός εισάγεται μόνο εκεί που χρειάζεται.

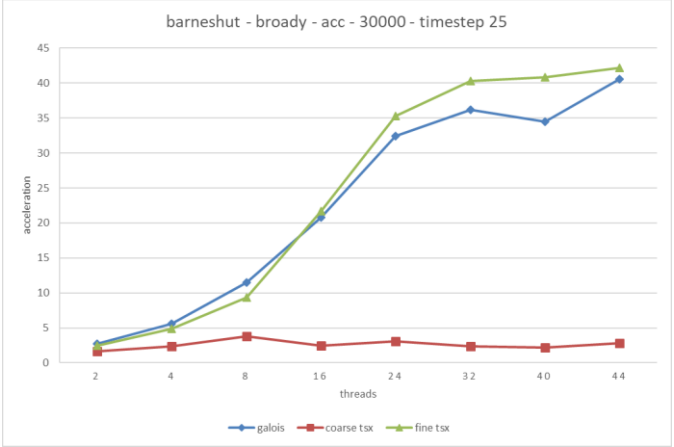

Figure 38: Σύγκριση επιτάχυνση της εκτέλεσης Barnes-Hut μεταξύ Galois, coarse grain tsx και fine grain tsx για 30000 ουράνια σώματα και 25 βήματα εκτέλεσης

Είδαμε ωστόσο ότι η υλοποίηση του συγχρονισμού μέσω transactional memory δεν είναι τόσο εύκολη όσο η εισαγωγή δύο transaction_start και transaction_end, καθώς λόγω περιορισμών στο hardware η εκτέλεση υποφέρει από πολλά capacity aborts. Απαιτείται λοιπόν διαφορετική προσέγγιση και μελέτη σε κάθε αλγόριθμο για να βγει το σωστό σχήμα συγχρονισμού.

Σε αντίθεση με τους barneshut και clustering αλγορίθμους ωστόσο, οι Delaunay triangulation και refinement υπέφεραν και από πολλά conflict aborts. Για να αντιμετωπιστεί

αυτό αρκεί να αυξηθεί ο αριθμός των προσπαθειών των transactions, καθώς το transaction με το οποίο υπάρχει σύγκρουση κάποια στιγμή θα τελειώσει την εκτέλεση του. Αυτό αν και ώθησε τον triangulation αλγόριθμο σε μικρότερους χρόνους, δεν ήταν αρκετό στον Delaunay refinement αλγόριθμο.

Στον refinement αλγόριθμο φάνηκε να προκαλείται πιθανώς live lock μεταξύ των εκτελέσεων των transaction, όπου το ένα προκαλούσε συνεχή abort στα άλλα. Η εισαγωγή μιας μικρής καθυστέρησης στην επανεκτέλεση ωθεί τον αλγόριθμο σε καλύτερους χρόνους, όπως επίσης και τον triangulation αλγόριθμο. Στους barneshut και clustering αλγορίθμους δεν είχε ιδιαίτερο αντίκτυπο η εισαγωγή καθυστέρησης, καθώς δεν εμφάνιζαν τον ίδιο αριθμό από conflict aborts. Ωστόσο, ο αριθμός της καθυστέρησης δεν μπορεί να είναι ίδιος για όλους τους αριθμούς νημάτων. Όσο περισσότερα νήματα χρησιμοποιεί ένα πρόγραμμα, τόσο μεγαλύτερη καθυστέρηση πρέπει να εισάγεται στην επανεκτέλεση. Προκύπτει λοιπόν η ανάγκη να υπάρχει ένα σύστημα που να βρίσκει δυναμικά αυτόν τον αριθμό, ή ακόμα και να υπάρχει υποστήριξη από το υλικό σχετικά με πληροφορίες για το αν η δοσοληψία που προκάλεσε σύγκρουση τρέχει ακόμα.

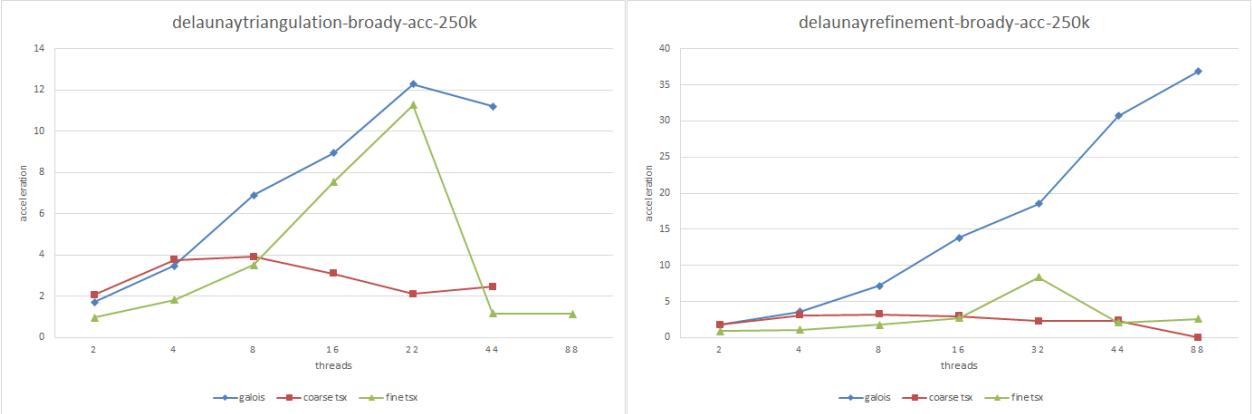

Figure 39: Σύγκριση επιτάχυνση της εκτέλεσης Del. Tri και Del. Ref μεταξύ Galois, coarse grain tsx και fine grain tsx για 250 χιλιάδες κόμβους

Επίσης, η απόδοση που πετύχαινε κάθε αλγόριθμος αύξανε καθώς αυξανόταν το μέγεθος του αρχικού input, το οποίο δείχνει ότι ο διαθέσιμος παραλληλισμός εξαρτάται και από το μέγεθος των διαθέσιμων δεδομένων. Τέλος, είδαμε ότι η απόδοση του Transactional memory χειροτερεύει αρκετά όταν υπάρχει Non Uniform Memory Access, ή όταν ενεργοποιείται το HyperThreading, με τους αλγορίθμους να αυξάνουν τα capacity aborts στις αντίστοιχες εκτελέσεις. Σε αρκετές περιπτώσεις φάνηκε διαφορά και στα capacity aborts μεταξύ του broady και haci3, πράγμα το οποίο δείχνει ότι το transactional memory θα υποφέρει αρκετά σε συστήματα με μικρό μέγεθος cache.

Ωστόσο, είναι προφανές ότι το Transactional Memory ως μια wait-free και lock-free τεχνική έχει αρκετά πλεονεκτήματα στην προγραμματιστική ευκολία και υλοποίηση σε σχέση με άλλες τεχνικές και μπορεί να χρησιμοποιηθεί για τον προγραμματισμό εφαρμογών γράφων. Η απόδοση, ανάλογα τον αλγόριθμο προς εξέταση, φτάνει στα επίπεδα μιας lock-free τεχνικής όπως αυτή του semantic commutativity. Ωστόσο, μια lock free software λύση για την υλοποίηση του συγχρονισμού θα αποδίδει καλύτερα στην αύξηση των νημάτων ιδιαίτερα στις NUMA περιοχές και στην χρήση HyperThreading.

7 Βιβλιογραφία

[1] W. D. Hills και G. L. S. Jr, «Data parallel algorithms,» *Communications of the ACM - Special issue on parallelism,* 1986.

[2] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, J. G. L. Steele και M. E. Zosel, The high performance Fortan handbook, Cambridge: MIT Press, 1994.

[3] «OpenMP,» [Ηλεκτρονικό]. Available: http://www.openmp.org/.

[4] «Go language Patterns,» Google, [Ηλεκτρονικό]. Available: http://www.golangpatterns.info/concurrency/parallel-for-loop.

[5] W. Pugh, «A practical algorithm for exact array dependence analysis,» *Communications of the ACM,* p. 114, 1992.

[6] M. Kulkarni, K. P. Patrick Carribault, G. Ramanarayanan, B. Walter, K. Bala και L. P. Chew, «Scheduling Strategies for Optimistic Parallel Execution ofIrregular Programs,» University of Texas; Cornell University, 2008.

[7] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala και L. P. Chew, «Optimistic Parallelism Benefits from Data Partitioning,» The University of Texas, Cornell University, 2008.

[8] B. Steensgaard, «Points-to Analysis in Almost Linear Time,» Microsoft Research, Redmond, WA 98052, USA, 1995.

[9] R. P. Wilson και M. S. Lam, «EfficientContext-SensitivePointerAnalysis for C programs,» Standford University, CA 94305, 1995.

[10] R. Ponnusam, J. Salt και A. Choudhary, «Runtime Compilation Techniques for DataPartitioning and Communication Schedule Reuse,» Syracuse Universit, 1993.

[11] L. G. Valiant, «A bridging model for parallel computation,» *Communications of the ACM,* p. 111, 1990.

[12] R. M. tomasulo, «An Efficient Algorithm for Exploiting Multiple Arithmetic Units,» 1967.

[13] M. Cintra, J. F. Martınez και o. Torrellas, «Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors,» University of Illinois, Urbana-Champaign, 2000.

[14] L. Hammond, M. Willey και K. Olukotun, «Data Speculation Support for a Chip Multiprocessor,» Stanford University.

[15] MauriceHerlihy και J. EliotB.Moss, «Transactional Memory: Architectural Support for Lock-Free Data Structures,» University of Massachusetts, Cambridge.

[16] A. Mcdonald, «Architectures for Transactional Memory,» Stanford University, 2009.

[17] C. S. course, «Implementing Transactional Memory,» Carnegie Mellon University, 2013.

[18] «TMWare,» [Ηλεκτρονικό]. Available: http://www.tmware.org/tinystm.

[19] R. Ennals, «Software Transactional Memory Should Not Be Obstruction Free,» Portland State University.

[20] T. Harris, S. Marlow, S. P. Jones και M. Herlihy, «Composable Memory Transactions,» Microsoft Research.

[21] D. Dice, O. Shalev και N. Shavit, «Transactional Locking II,» Sun Microsystems Laboratories, 2006.

[22] M. L. Scott, *The University of Rochester STM.*

[23] T. Harris, M. Plesko, A. Shinnar και D. Tarditi, «Optimizing Memory Transactions,» Microsoft Research, Harvard University, 2006.

[24] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh και B. Hertzberg, «A high performance software transactional memory system for a multi-core runtime,» *PPoPP '06,* p. 197, 2006.

[25] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis και K. Olukotun, «Transactional Memory Coherence and Consistency,» Stanford University, 2004.

[26] A. Shriraman, S. Dwarkadas και M. L. Scott, «Flexible Decoupled Transactional Memory Support,» University of Rochester, 2008.

[27] L. Ceze, J. Tuck, C. Cascaval και o. Torrellas, «Bulk Disambiguation of Speculative Threads in Multiprocessors,» University of Illinois, 206.

[28] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson και S. Lie, «Unbounded Transactional Memory,» MIT Computer Science and Artificial Intelligence Laboratory, 2005.

[29] R. Rajwar, M. Herlihy και K. Lai, «Virtualizing Transactional Memory,» Intel Corporation, Brown University, 2005.

[30] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill και D. A. Wood, «LogTM: Log-based Transactional Memory,» University of Wisconsin–Madison, 2006.

[31] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari και E. Witche, «MetaTM/TxLinux: Transactional Memory For An Operating,» University of Texas at Austin, 2007.

[32] H. Avni και T. Brown, «PHyTM: Persistent Hybrid Transactional Memory,» European Research Institute, University of Toronto, 2016.

[33] I. Calciu, J. Gottschlich, T. Shpeisman, G. Pokam και M. Herlihy, «Invyswell: A Hybrid Transactional Memory for Haswell'sRestricted Transactional Memory,» Brown University, Intel Labs, 2014.

[34] D. Kanter, «Real World Technologies,» 2012. [Ηλεκτρονικό]. Available: Analysis of Haswell's Transactional Memory.

[35] «Intel® 64 and IA-32 Architectures Optimization Reference Manual,» Intel Corporation, June 2016.

[36] «Intel® 64 and IA-32 Architectures Software Developer's Manual,» Intel Corporation, September 2016.

[37] J. B. C. Neto, P. A. Wawrzynek, M. T. M. Carvalho, L. F. Martha και A. R. Ingraffea, «An Algorithm for Three-Dimensional Mesh Generation for Arbitrary Regions with Cracks,» Springer-Verlag, London, 2001.

[38] B. Selman, H. Levesque και D. Mitchell, «A New Method for Solving Hard Satisfiability Problems,» Proceedings of the Tenth National Conference on Artificial Intelligence, 1992.

[39] «SETL,» https://en.wikipedia.org/wiki/SETL.

[40] A. Bernstein, «Analysis of programs for parallel processing,» IEEE Transactions on Electronic Computers, 1966.

[41] A. Rogers και K. Pingali, «Process decomposition through locality of reference,» PLDI '89 Proceedings of the ACM SIGPLAN 1989 conference on Programming language design and implementation, 1989.

[42] W. N. S. III και M. L. Scott, «Advanced Contention Management for Dynamic Software Transactional Memory,» University of Rochester, 2005.

[43] P. H. J. Barnes, «A hierarchical O(N log N) force-calculation algorithm,» Nature, 1986.