ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Μελέτη και Αξιολόγηση Τεχνικών Παραλληλοποίησης Δομών Δεδομένων και Αλγορίθμων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Χριστίνα Χρ. Γιαννούλα

Επιβλέπων: Γεώργιος Γκούμας
Λέκτορας Ε.Μ.Π.

Αθήνα, Ιούλιος 2016

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑ-ΤΩΝ

Μελέτη και Αξιολόγηση Τεχνικών Παραλληλοποίησης Δομών Δεδομένων και Αλγορίθμων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Χριστίνα Χρ. Γιαννούλα

Επιβλέπων: Γεώργιος Γκούμας
Λέκτορας Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 14η Ιουλίου 2016.

....	....	....
Γ. Γκούμας	Ν. Κοζύρης	Κ. Σαγώνας
Λέκτορας Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Αναπληρωτής Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2016.

....
Χριστίνα Χρ. Γιαννούλα
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © Χριστίνα Γιαννούλα, 2016. Εθνικό Μετσόβιο Πολυτεχνείο.
Με επιφύλαξη κάθε δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ' ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπο την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τη συγγραφέα.
Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τη συγγραφέα και δεν πρέπει να ερμηνευτεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Στις μέρες μας, οι πολυπύρηνοι επεξεργαστές έχουν γίνει η κυρίαρχη πλατφόρμα υπολογισμών και έχουν εισαχθεί σε πολλά προγραμματιστικά περιβάλλοντα. Ο παράλληλος προγραμματισμος δεν αφορά πλέον μόνο επιστημονικές εφαρμογές που τρέχουν σε υπερυπολογιστές, αλλα καλύπτει επίσης ένα μεγάλο φάσμα εφαρμογών για προσωπικούς υπολογιστές. Ένα από τα πιο δύσκολα προβλήματα στα συστήματα παράλληλης επεξεργασίας είναι η ανάπτυξη παράλληλου λογισμικού το οποίο κλιμακώνει αποδοτικά. Αρκετές εφαρμογές δεν κλιμακώνουν μετά από έναν αριθμό επεξεργαστών εξαιτίας του αυξημένου κόστους επικοινωνίας. Προκειμένου να αξιοποιηθούν οι διαθέσιμες αρχιτεκτονικές, οι βασικές δομές δεδομένων και οι σειριακοί αλγόριθμοι πρέπει να επανασχεδιασθούν. Το πρώτο μέρος αυτής της διπλωματικής αφορά τις παράλληλες δομές δεδομένων, με ιδιαίτερη έμφαση στα δυαδικά δέντρα αναζήτησης, εξετάζοντας τον τρόπο συγχρονισμού τους, τα ιδιαίτερα χαρακτηριστικά τους και την κλιμακωσιμότητα που προσφέρουν. Στο δεύτερο μέρος της διπλωματικής παρουσιάζεται μια παραλληλοποίηση του αλγορίθμου του Dijkstra που είναι ένας σειριακός αλγόριθμος. Αυτή η υλοποίηση χρησιμοποιεί Transactional Memory, για να συντονίσει αποτελεσματικά τις ταυτόχρονες προσβάσεις των νημάτων στις κοινές δομές δεδομένων και την έννοια των Helper Threads, για να εξάγει παραλληλισμό. Η αξιολόγηση του αλγορίθμου γίνεται σε ένα σύστημα που υποστηρίζει Hardware Transactional Memory.

Λέξεις-Κλειδία: παράλληλες δομές δεδομένων, δυαδικά δέντρα αναζήτησης, κλιμακωσιμότητα, παράλληλος προγραμματισμός, αλγόριθμος του Dijkstra, Helper Threads, Hardware Transactional Memory

Abstract

Nowadays, multicore processors have become the dominant computing platform and are being used by many programming environments. Parallel programming is no longer about scientific applications run in supercomputers, but covers a wider range of applications on personal computers, too. The most difficult problem is to develop parallel software that scales efficiently. Several applications do not scale further than a number of processors due to communication overhead. To exploit the available architectures basic data structures and sequential algorithms must be redesigned. In the first part of this thesis we study concurrent data structures, particularly focusing on concurrent binary search trees, with respect to the way they are synchronized, their special characteristics and the scalability they provide. The second part of this thesis presents a parallelization of the inherently serial Dijkstra's algorithm. This implementation employs Transactional Memory to efficiently orchestrate the concurrent thread's accesses to shared data structures and the concept of Helper Threads to extract parallelism. We evaluate the execution of the algorithm on a system that supports Hardware Transactional Memory.

Keywords: Concurrent Data Structures, Binary Search Trees, scalability, parallel programming, Dijkstra's algorithm, Helper Threads, Hardware Transactional Memory

Ευχαριστίες

Η παρούσα διπλωματική εργασία εκπονήθηκε στο Εργαστήριο Υπολογιστικών Συστημάτων της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου, υπό την επίβλεψη του Λέκτορα Ε.Μ.Π. Γεώργιου Γκούμα.

Θα ήθελα να ευχαριστήσω τον καθηγητή μου κ. Γεώργιο Γκούμα, για την εποπτεία κατά την εκπόνηση της εργασίας μου, τις γνώσεις που μου προσέφερε με τη διδασκαλία του και την ευκαιρία που μου έδωσε να δουλέψω στο εργαστήριο.

Θα ήθελα επίσης να ευχαριστήσω τον καθηγητή της σχολής κ. Νεκτάριο Κοζύρη για την έμπνευση που μου προσέφερε με τη διδασκαλία του και να τον συγχαρώ για υψηλό επίπεδο σπουδών των μαθημάτων διδασκαλίας του και του εργαστηρίου.

Ιδιαίτερα θα ήθελα να ευχαριστήσω τον Υποψήφιο Διδάκτωρ Δημήτριο Σιακαβάρα για τη συνεχή καθοδήγησή του κατά τη διάρκεια εκπόνησης αυτής της διπλωματικής εργασίας, καθώς και για την ενθάρρυνσή του, την υπομονή του και το χρόνο που αφιέρωσε. Χωρίς τη συμβολή του η ολοκλήρωση αυτής της εργασίας δε θα ήταν εφικτή.

Επιπλέον, επιβάλλεται να ευχαριστήσω τους φίλους μου και τους συμφοιτητές μου για την πολύτιμη βοήθειά τους σε επιστημονικό και προσωπικό επίπεδο.

Τέλος, θα ήθελα να ευχαριστήσω θερμά την οικογένειά μου, τους γονείς μου και την αδερφή μου για την αγάπη, την αμέριστη υποστήριξή τους όλα αυτά τα χρόνια και την εμπιστοσύνη τους σε κάθε μου επιλογή.

Χριστίνα Γιαννούλα

Περιεχόμενα

1 Εισαγωγή **19**
1.1 Επισκόπηση 19
1.2 Ο νόμος του Amdahl 20
1.3 Παράλληλες αρχιτεκτονικές 22
1.3.1 Αρχιτεκτονική κοινής μνήμης 22
1.3.2 Αρχιτεκτονική κατανεμημένης μνήμης 24
1.3.3 Υβριδική αρχιτεκτονική μνήμης 25
1.4 Συγχρονισμός 26

2 Παράλληλα δυαδικά δέντρα αναζήτησης **29**
2.1 Παράλληλες δομές δεδομένων 29
2.2 Δέντρα αναζήτησης 30
2.2.1 Απλά δυαδικά δέντρα αναζήτησης (BST) 30
2.2.2 AVL Δέντρα 31
2.2.3 Red-Black Δέντρα 32
2.3 Τεχνικές κατασκευής παράλληλων δομών δεδομένων 33
2.3.1 Coarse-grained locking 34
2.3.2 Fine-grained locking 34
2.3.3 Lock-free προγραμματισμός 34
2.4 Βασικές λειτουργές παράλληλων δέντρων αναζήτησης 35
2.5 Μία απλοϊκή προσέγγιση 35
2.5.1 Περιγραφή 36
2.5.2 Λεπτομέρειες υλοποίησης 37
2.6 Μία πιο πολύπλοκη προσέγγιση 39
2.6.1 Περιγραφή 39
2.6.2 Λεπτομέρειες υλοποίησης 42
2.7 Αξιολόγηση 42
2.7.1 Χαρακτηριστικά συστήματος αξιολόγησης 42
2.7.2 Χαρακτηριστικά εκτέλεσης 43
2.7.3 Αποτελέσματα 43

3 Transactional Memory **51**
3.1 Transactional Memory (TM) . . . 51
3.1.1 Software Transactional Memory (STM) . . . 52
3.1.2 Hardware Transactional Memory (HTM) . . . 52
3.1.3 Hybrid Transactional Memory . . . 52
3.2 Βασικά χαρακτηριστικά TM συστημάτων . . . 53
3.3 Intel's Haswell HTM . . . 53
3.3.1 Transactional Synchronizations Extensions (TSX) . . . 54

4 Παραλληλοποιώντας τον αλγόριθμο του Dijkstra **59**
4.1 Ο αλγόριθμος του Dijkstra . . . 59
4.1.1 Ο αλγορίθμος . . . 59
4.1.2 Η πολυπλοκότητα του αλγορίθμου . . . 61
4.2 Μία τεχνική παραλληλοποίησης στον αλγόριθμο του Dijkstra . . . 61
4.3 Χαρακτηριστικά συστήματος . . . 66
4.4 Αξιολόγηση . . . 67
4.4.1 Αξιολόγηση του σειριακού αλγορίθμου . . . 67
4.4.2 Αξιολόγηση του παράλληλου αλγορίθμου . . . 70
4.5 Αποτελέσματα . . . 76
4.5.1 Η απόδοση του αλγορίθμου . . . 76
4.5.2 Αναλύοντας περισσότερο τα αποτελέσματα . . . 77
4.5.3 Αξιολόγηση της τεχνικής structure padding . . . 80
4.6 Χρησιμοποιώντας skip list . . . 82
4.6.1 Η skip list δομή . . . 82
4.6.2 Σύγκριση με το δυαδικό σωρό . . . 85
4.6.3 Αποτελέσματα . . . 88

5 Συμπεράσματα και Μελλοντικές Επεκτάσεις **93**

Κατάλογος Σχημάτων

1.1 Η συνολική επιτάχυνση ενός παράλληλου προγράμματος, καθώς το παράλληλο κλάσμα του προγράμματος και ο αριθμός των επεξεργαστών αλλάζουν. 21
1.2 Η ταξινόμηση του Flynn. 23
1.3 Κλασσική οργάνωση μιας SMP αρχιτεκτονικής. 24
1.4 Κλασσική οργάνωση μνήμης σε μια αρχιτεκτονική κατανεμημένης μνήμης. 25
1.5 Κλασσική οργάνωση μια Υβριδικής αρχιτεκτονικής. 26

2.1 Το intreface μιας δομής αναζήτησης. Οι λειτουργίες ενημέρωσης έχουν δύο φάσεις: μία φάση αναζήτησης του στοιχείου και μία φάση εκτέλεσης της ενημέρωσης στη δομή. 30
2.2 Ένα παράδειγμα ενός external και ενός internal δέντρου. 31
2.3 Μία δεξιά και μία αριστερή περιστροφή δέντρου. 32
2.4 Ένα παράδειγμα ενός red–black δέντρου. 33
2.5 Μία αναπαράσταση ενός bst κόμβου στη μνήμη χρησιμοποιώντας padding, έτσι ώστε αυτός να καταλαμβάνει ακριβώς μία cache line. 38
2.6 Κατά την διαγραφή ενός κόμβου D σε ένα internal δέντρο, πρέπει το υπόδεντρο με ρίζα τον κόμβο D να παραμένει κλειδωμένο. 39
2.7 Η απόδοση απλοϊκών παράλληλων δυαδικών δέντρων για εύρος κλειδιών 2K και για 3 διαφορετικές αναλογίες λειτουργιών. 44
2.8 Η απόδοση απλοϊκών παράλληλων δυαδικών δέντρων για εύρος κλειδιών 32K και για 3 διαφορετικές αναλογίες λειτουργιών. 45
2.9 Η απόδοση απλοϊκών παράλληλων δυαδικών δέντρων για εύρος κλειδιών 2000000 και για 3 διαφορετικές αναλογίες λειτουργιών. 46
2.10 Η απόδοση των πολύπολοκων παράλληλων δυαδικών δέντρων για εύρος κλειδιών 2K και για 3 διαφορετικές αναλογίες λειτουργιών. 47
2.11 Η απόδοση των πολύπολοκων παράλληλων δυαδικών δέντρων για εύρος κλειδιών 32K και για 3 διαφορετικές αναλογίες λειτουργιών. 48
2.12 Η απόδοση των πολύπολοκων παράλληλων δυαδικών δέντρων για εύρος κλειδιών 2000000 και για 3 διαφορετικές αναλογίες λειτουργιών. 48

3.1 Η κατάσταση της δοσοληψίας αποτυπώνεται στα bit του EAX καταχωρητή. 56
3.2 Μία παράλληλη εκτέλεση δύο νημάτων με RTM που οδηγεί σε προβλήματα συνάφειας της μνήμης. 58

4.1 Σχήμα εκτέλεσης του αλγορίθμου. 62
4.2 Αξιολογήση των 4 φάσεων του σειριακού αλγορίθμου σε 3 διαφορετικούς γράφους. 68
4.3 Αξιολόγηση της τεχνικής structure padding στο σειριακό αλγόριθμο. . . . 69
4.4 Ο χρόνος εκτέλεσης για έναν random node-1M-edge-10M γράφο για διαφορετικό αριθμό επαναλήψεων ανά δοσοληψία του main thread. 70
4.5 Ο χρόνος εκτέλεσης για έναν rmat node-10M-edge-500M γράφο για διαφορετικό αριθμό επαναλήψεων ανά δοσοληψία του main thread. 71
4.6 Ο χρόνος εκτέλεσης στον random node-1M-edge-10M γράφο για διαφορετικό αριθμό επαναλήψεων ανά δοσοληψία των helper threads. 72
4.7 Ο χρόνος εκτέλεσης σε ένα rmat node-10M-edge-500M γράφο για διαφορετικό αριθμό δυνατων επαναλήψεων ανά δοσοληψία των helper threads. 73
4.8 Ο χρόνος εκτέλεσης για ένα γράφο random node-1M-edge-100M για διαφορετικό αριθμό γειτόνων προς εξέταση για relaxation σε μία δοσοληψία του main thread. 74
4.9 Ο χρόνος εκτέλεσης για ένα γράφο rmat node-10M-edge-500M για διαφορετικό αριθμό γειτόνων προς εξέταση για relaxation σε μια δοσοληψία του main thread. 74
4.10 Ο χρόνος εκτέλεσης για ένα γράφο random node-1M-edge-100M για διαφορετικό αριθμό γειτόνων προς εξέταση για relaxation σε μια δοσοληψία των helper threads. 75
4.11 Ο χρόνος εκτέλεσης για ένα γράφο rmat node-10M-edge-500M για διαφορετικό αριθμό γειτόνων προς εξέταση για relaxation σε μια δοσοληψία των helper threads. 75
4.12 Η απόδοση του αλγορίθμου για διαφορετικό πλήθος νημάτων για γράφους διαφορετικής πυκνότητας. 76
4.13 Κατανομή των relaxations ανάμεσα στο main και τα helper threads. . . . 78
4.14 Ο αριθμός των commits/aborts του main thread. 79
4.15 Το ποσοστό των συνολικών transactional aborts για όλα τα νήματα στο συνολικό αριθμό δοσοληψιών. 80
4.16 Κατανομή του χρόνου στις διάφορες φάσεις εκτέλεσης του main thread. . 81
4.17 Παρά το ότι δε χρησιμοποιήθηκε padding, ταυτόχρονα relaxations εκτελούνται σε κορυφές που βρίσκονται σε διαφορετικές cache lines. 82
4.18 Αξιολόγηση της τεχνικής padding στον παράλληλο αλγόριθμο. 83
4.19 Ένα παράδειγμα μιας skip list. 84
4.20 Ο χρόνος εκτέλεσης του σειριακού αλγορίθμου για τις 3 διαφορετικές δομές που χρησιμοποιούνται για την ουρά προτεραιότητας. 86
4.21 Ο χρόνος εκτέλεσης της παράλληλου αλγορίθμου για τις 3 διαφορετικές δομές που χρησιμοποιούνται για την ουρά προτεραιότητας. 87
4.22 Η κατανομή των relaxations ανάμεσα στο main και τα helper threads για τον random-node-10M-edge-500M γράφο για 2 διαφορετικές δομές που χρησιμοποιούνται για την ουρά προτεραιότητας. 87

4.23 Ο αριθμός των commits/aborts του main thread στον παράλληλο αλγόριθμο για τις 3 διαφορετικές δομές που χρησιμοποιούνται για την ουρά προτεραιότητας. 89
4.24 Ο αριθμός των commits/aborts των helper threads για τον παράλληλο αλγόριθμο για τις 3 διαφορετικές δομές που χρησιμοποιούνται για την ουρά προτεραιότητας. 89
4.25 Η απόδοση του αλγορίθμου για γράφους διαφορετικής πυκνότητας με χρήση της βελτιστοποιημένης skip list. 90
4.26 Ένα παράδειγμα binary heap. Οι κορυφές με κόκκινο χρώμα δεν έχουν ακόμα αποκτήσει τις βέλτιστες τιμές τους. 91
4.27 Μια προσέγγιση της απόδοσης που βασίζεται στα relaxations που εκτελεί το main thread στην περίπτωση των 14 νημάτων και η απόδοση που επετεύχθει όταν χρησιμοποιήθηκε binary heap. 91
4.28 Μια προσέγγιση της απόδοσης που βασίζεται στα relaxations που εκτελεί το main thread στην περίπτωση των 14 νημάτων και η απόδοση που επετεύχθει όταν χρησιμοποιήθηκε η βελτιστοποιημένη skip list. 92

Listings

2.1 Μία τυπική δομή bst κόμβου 38
3.1 Παράδειγμα χρήσης HLE 55
3.2 Παράδειγμα: προσθήκη του καθολικού κλειδώματος στο read set της δοσοληψίας 57
3.3 Ένα RTM παράδειγμα 57
4.1 Ο αλγόριθμος του Dijsktra 60
4.2 Ο κώδικας του main thread για ένα πραγματικό HTM σύστημα. 65
4.3 Ο κώδικας των helper threads για ένα πραγματικό HTM σύστημα. 65
4.4 Οι 4 φάσεις του αλγορίθμου. 67

Κεφάλαιο 1

Εισαγωγή

1.1 Επισκόπηση

Το 1965, ο Moore προέβλεψε ότι ο αριθμός των τρανζίστορ σε ένα ολοκληρωμένο κύκλωμα θα διπλασιάζεται σχεδόν κάθε 2 χρόνια. Ωστόσο, η αύξηση του αριθμού των τρανζίστορ ανά επεξεργαστή θέτει κάποιους φυσικούς περιορισμούς. Τα πολύ πυκνά τσιπ χρησιμοποιούν περισσότερη ηλεκτρική ενέργεια και παράγουν περισσότερη θερμότητα περιορίζοντας την εξέλιξη των επεξεργαστών.

Ο νόμος του Moore χρησιμοποιήθηκε στη βιομηχανία ημιαγωγών. Ολοένα και περισσότερα τραζίστορ ενσωματώθηκαν στο ίδιο τσιπ και η απόδοση διπλασιαζόταν κάθε 18 μήνες. Αυτοί οι υψηλής απόδοσης μικροεπεξεργαστές ονομάστηκαν πολυπύρηνοι επεξεργαστές. Σήμερα, οι πολυπύρηνοι επεξεργαστές αποτελούν το βασικό συστατικό υπερυπολογιστών, κατανεμημένων συστημάτων και προσωπικών υπολογιστών. Η αύξηση του αριθμού των επεξεργαστών δίνει πολύ υψηλή απόδοση κατά την εκτέλεση υπολογισμών. Όμως, τα πολυπύρηνα συστήματα προκαλούν προβλήματα επικοινωνίας και συγχρονισμού και η κλιμακωσιμότητα αυτών των συστημάτων αποτελεί μεγάλη πρόκληση.

Προκειμένου να εκμεταλλευτεί όλες αυτές τις διαθέσιμες πηγές υλικού, η βιομηχανία των υπολογιστών ανέπτυξε νέες αρχιτεκτονικές. Για παράδειγμα, η χρήση βαθύτερων superscalar αρχιτεκτονικών βελτίωσε την εκτέλεση μίας εντολής και ως συνέπεια οι συμβατικές αρχιτεκτονικές αντικαταστάθηκαν με παράλληλες αρχιτεκτονικές με σκοπό να μεγιστοποιήσουν την απόδοση των εφαρμογών.

Τα πολυπύρηνα συστήματα αποτελούν πλέον μία λύση σε εφαρμογές απαιτητικές σε υπολογισμούς. Επιστημονικές εφαρμογές όπως προσομοιώσεις αστροφυσικών φαινομένων, εφαρμογές για πρόβλεψη καιρού, εφαρμογές στη βιομηχανία αλλά και σε άλλους τομείς καθημερινής χρήσης όπως οι μηχανές αναζήτησης είναι υπολογιστικά απαιτητικές. Οι σύγχρονοι σειριακοί αλγόριθμοι, αν και βελτιστοποιημένοι, δε μπορούν να προσφέρουν τη βέλτιστη απόδοση όταν εκτελούνται σε πολυπύρηνες αρχιτεκτονικές. Επομένως, οι σειριακοί αλγόριθμοι πρέπει να επανασχεδιασθούν, έτσι ώστε να τρέχουν παράλληλα και να εκμεταλλευτούν τις διαθέσιμες αρχιτεκτονικές.

Ο παράλληλος προγραμματισμός απαιτεί συγχρονισμό μεταξύ των παράλληλων διερ-

γασιών με σκοπό να αποφύγει τις συγκρούσεις και τις καταστάσεις συναγωνισμού. Τεχνικές συγχρονισμού υλοποιούνται τόσο σε software όσο και σε hardware. Ωστόσο, η επικοινωνία και ο συγχρονισμός των διαφόρων εργασιών είναι τα μεγαλύτερα εμπόδια στην επίτευξη υψηλής απόδοσης σε παράλληλες εφαρμογές. Η πρόκληση είναι να δημιουργηθεί hardware και software μέσα από τα οποία θα καθίσταται εύκολη η ανάπτυξη παράλληλων προγραμμάτων που προσφέρουν υψηλή απόδοση και κλιμακωσιμότητα καθώς ο αριθμός των πυρήνων ανά τσιπ αυξάνει.

1.2 Ο νόμος του Amdahl

Στους παράλληλους αλγορίθμους, ένα μεγάλο πρόβλημα χωρίζεται σε μικρότερα, τα οποία επιλύονται ταυτόχρονα. Κάθε διαθέσιμος πυρήνας αναλαμβάνει την εκτέλεση ενός μικρότερου προβλήματος. Συγκριτικά με τους σειριακούς αλγόριθμους, είναι πιο δύσκολο να αναπτυχθεί ένας παράλληλος αλγόριθμος. Ωστόσο ένα παράλληλο πρόγραμμα έχει καλύτερη απόδοση και κλιμακωσιμότητα.

Ο νόμος του Amdahl [1] είναι ένας θεωρητικός τύπος που δίνει τη μέγιστη θεωρητική επιτάχυνση (speedup) που μπορεί να επιτευχθεί. Η επιτάχυνση είναι ένα μέγεθος που δείχνει πόσες φορές το παράλληλο πρόγραμμα είναι γρηγορότερο από τον καλύτερο σειριακό αλγόριθμο. Αν T_s είναι ο χρόνος εκτέλεσης του καλύτερου σειριακού αλγορίθμου και T_p είναι ο χρόνος εκτέλεσης μιας παράλληλης εκδοχής του προγράμματος σε p επεξεργαστές, η επιτάχυνση ορίζεται ως:

$$Speedup(S) = \frac{T_s}{T_p}$$

Τυπικά, η επιτάχυνση S σχετίζεται με τον αριθμό των επεξεργαστών p με την ανισότητα: $S \leq p$. Αν $S = p$, η επιτάχυνση είναι *γραμμική*.

Ο νόμος του Amdahl δείχνει ότι η θεωρητική επιτάχυνση αυξάνεται βελτιώνοντας ένα κομμάτι του προγράμματος. Ας θεωρήσουμε ότι f είναι το κλάσμα του προβλήματος το οποίο δεν μπορεί να παραλληλοποιηθεί και πρέπει να εκτελεστεί σειριακά, τότε ο χρόνος εκτέλεσης ενός παράλληλου προγράμματος είναι:

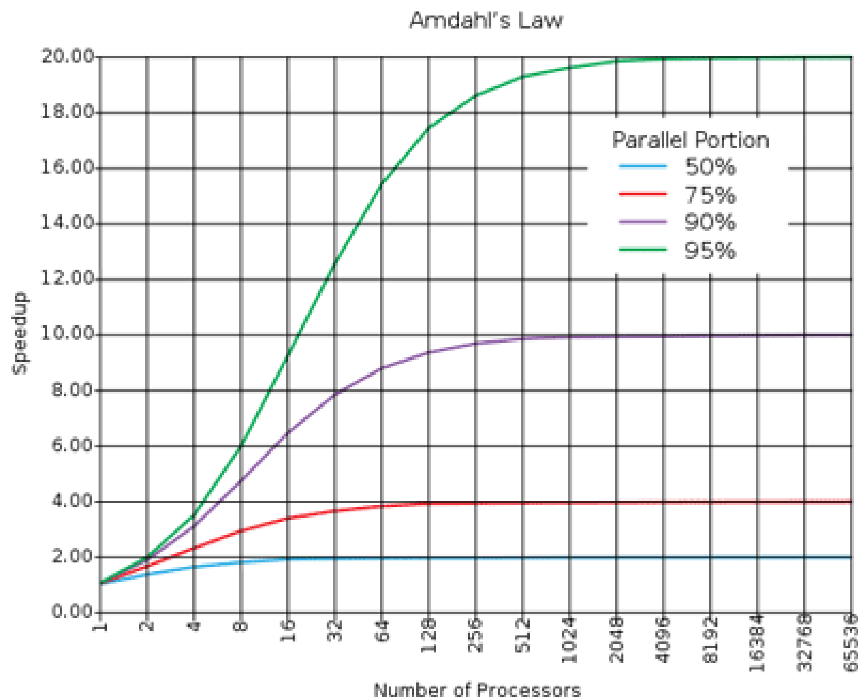
$$T_p = fT_s + \frac{(1-f)T_s}{p}$$

και η έκφραση για την επιτάχυνση γίνεται:

$$\text{Συνολική Επιτάχυνση (speedup)s } S = \frac{1}{f + \frac{1-f}{p}}$$

Ο νόμος του Amdahl χρησιμοποιείται συχνά για να προβλέψει τη θεωρητική επιτάχυνση που μπορεί να επιτευχθεί όταν χρησιμοποιούνται περισσότεροι του ενός επεξεργαστές. Όταν ο αριθμός των επεξεργαστών p τείνει να γίνει άπειρος, η συνολική επιτάχυνση τείνει στο $1/f$. Δηλαδή, η θεωρητική επιτάχυνση περιορίζεται από το κομμάτι του προγράμματος που δε μπορεί να εκτελεσθεί παράλληλα. Αν το παράλληλο κομμάτι του προγράμματος είναι σχετικά μικρό, η επιτάχυνση θα είναι αντίστοιχα μικρή. Για παράδειγμα, αν

το σειριακό κλάσμα είναι $f = 90\%$, τότε το παράλληλο πρόγραμμα μπορεί να είναι 10 φορές γρηγορότερο στην καλύτερη περίπτωση από το σειριακό πρόγραμμα, ανεξάρτητα από τον αριθμό των επεξεργαστών. Η εικόνα 1.1 απεικονίζει τη συνολική επιτάχυνση παράλληλων εκτελέσεων για διαφορετικά σειριακά κλάσματα f και αριθμό επεξεργαστών p. Σύμφωνα με αυτήν, η συνολική επιτάχυνση του προγράμματος περιορίζεται από το σειριακό μέρος του προγράμματος και η χρήση περισσοτέρων επεξεργαστών δεν αυξάνει την επιτάχυνση σε όλες τις περιπτώσεις. Η παραλληλοποίηση μεγαλύτερου μέρους του σειριακού προγράμματος είναι η λύση για να μεγιστοποιήσουμε την απόδοση.



Σχήμα 1.1: Η συνολική επιτάχυνση ενός παράλληλου προγράμματος, καθώς το παράλληλο κλάσμα του προγράμματος και ο αριθμός των επεξεργαστών αλλάζουν.

Η κλιμακωσιμότητα, είναι ένα άλλο μέγεθος μέτρησης της απόδοσης που αναφέρεται στην ικανότητα ενός συστήματος ή ενός προγράμματος να αυξήσουν την απόδοση τους όταν ο αριθμός των επεξεργαστών αυξηθεί. Ας υποθέσουμε ότι το πρόγραμμα έχει ένα σταθερό μέγεθος. Ο χρόνος εκτέλεσης του προγράμματος αναμένεται να κλιμακώνει καθώς ο αριθμός των επεξεργαστών αυξάνεται. Παρ'όλα αυτά, υπάρχουν πολλοί παράγοντες που περιορίζουν την κλιμακωσιμότητα ενός προγράμματος. Πρώτον, το προγραμμα πρέπει να χωρίζεται σε μικρά ίσα κομμάτια κώδικα, καθένα από τα οποία πρέπει να εκτελείται σε διαφορετικό επεξεργαστή. Αν αυτά τα κομμάτια κώδικα δεν είναι ίσα, κάποιοι επεξεργαστές θα περιμένουν αλλους με μεγαλύτερα να τερματίσουν. Επιπλέον, η κλιμακωσιμότητα περιορίζεται και εξαιτίας του χρόνου που σπαταλάται στην επικοινωνία και

στο συγχρονισμό μεταξύ των επεξεργαστών. Αν ο χρόνος συγχρονισμού και επικοινωνίας είναι συγκρίσιμος με το συνολικό χρόνο εκτέλεσης, το πρόγραμμα δε θα κλιμακώνει με την αύξηση του αριθμού των επεξεργαστών. Συνοψίζοντας η κατανομή φόρτου εργασίας στους επεξεργαστές και ο χρόνος που ξοδεύεται για συγχρονισμό και επικοινωνία θέτουν σημαντικούς περιορισμούς στην κλιμακώσιμότητα ενός προγράμματος και πρέπει να λαμβάνονται υπόψιν στον παράλληλο προγραμματισμό.

1.3 Παράλληλες αρχιτεκτονικές

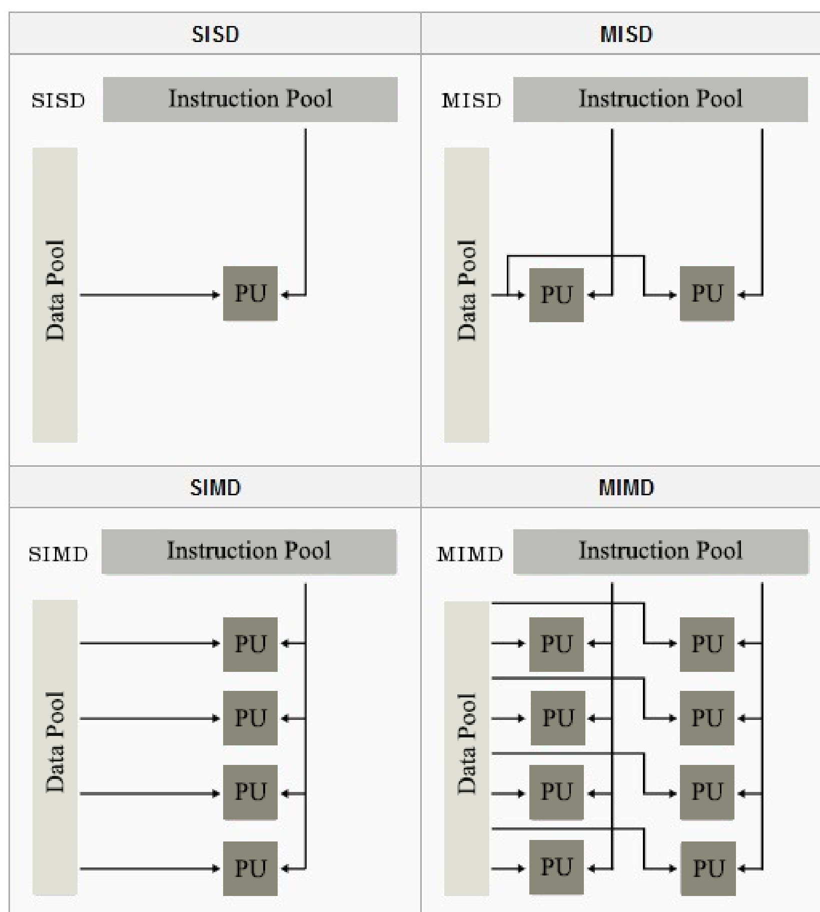
Η ταξινόμηση του Flynn διαχωρίζει τις αρχιτεκτονικές υπολογιστών, σύμφωνα με το επίπεδο του παραλληλισμού που χρησιμοποιούν για να επεξεργαστούν τις ροές εντολών και δεδομένων. Υπάρχουν 4 κατηγορίες:

- **SISD:** Single Instruction, Single Data
  Ένας σειριακός υπολογιστής. Οι σειριακοί υπολογιστές δεν εκτελούν εντολές παράλληλα.
- **SIMD:** Single Instruction, Multiple Data
  Ένας παράλληλος υπολογιστής με μία ροή εντολών, η οποία εκτελεί την ίδια εντολή σε πολλαπλά δεδομένα.
- **MISD:** Multiple Instruction, Single Data
  Πολλαπλές μονάδες επεξεργασίας εκτελούν περισσότερες εντολές στα ίδια δεδομένα. Η MISD αρχιτεκτονική δεν χρησιμοποιείται στο εμπόριο.
- **MIMD:** Multiple Instruction, Multiple Data
  Ένας παράλληλος υπολογιστής, στον οποίο κάθε επεξεργαστής εκτελεί ανεξάρτητες ροές εντολών σε ανεξάρτητα δεδομένα. Αυτή η αρχιτεκτόνική είναι η πιο συχνά και ευρέως χρησιμοποιημένη παράλληλη αρχιτεκτονική. Δύο παραδείγματα τέτοιας αρχιτεκτονικής είναι τα clusters και τα συστήματα πολυπύρηνων αρχιτεκτονικών.

Όπως αναφέρθηκε προηγουμένως, η MIMD πολυεπεξεργαστική αρχιτεκτονική είναι η πιο διαδεδομένη παράλληλη αρχιτεκτονική και κατάλληλη για μια πληθώρα εφαρμογών. Οι MIMD αρχιτεκτονικές μπορούν να κατηγοριοποιηθούν με βάση την οργάνωση της μνήμης τους σε 3 κατηγορίες: αρχιτεκτονικές κοινής μνήμης, αρχιτεκτονικές κατανεμημένης μνήμης και υβριδικές αρχιτεκτονικές.

1.3.1 Αρχιτεκτονική κοινής μνήμης

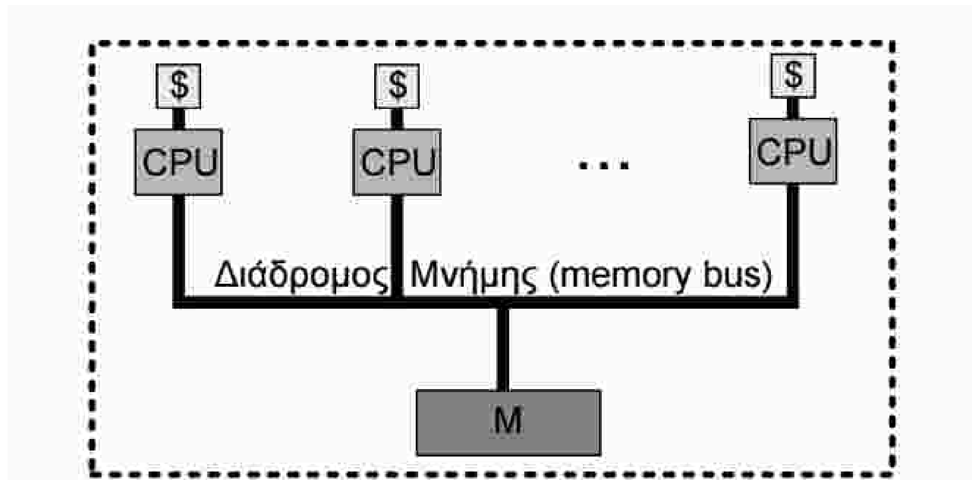
Στις αρχιτεκτονικές κοινής μνήμης, κάθε επεξεργαστής έχει τη δική του ιδιωτική κρυφή μνήμη και όλοι οι επεξεργαστές μοιράζονται ένα φυσικό χώρο, γνωστό ως καθολική μνήμη. Ένα σύστημα διαύλου διασυνδέει όλους τους επεξεργαστές. Τέτοια συστήματα μπορούν να εκτελούν ανεξάρτητες εργασίες, οι οποίες έχουν το δικό τους εικονικό



Σχήμα 1.2: Η ταξινόμηση του Flynn.

χώρο διευθύνσεων, ακόμα και αν μοιράζονται τον ίδιο φυσικό χώρο διευθύνσεων. Οι επεξεργαστές επικοινωνούν με κοινές μεταβλητές αποθηκευμένες στην κοινή καθολική μνήμη και μπορούν να κάνουν πρόσβαση σε οποιαδήποτε διεύθυνση μνήμης μέσω loads και stores. Στην περίπτωση που η πρόσβαση σε μία διεύθυνση μνήμης διαρκεί τον ίδιο χρόνο για όλους τους επεξεργαστές, η οργάνωση της μνήμης είναι συμμετρική (Symmetric multiprocessor SMP) και παρουσιάζεται στην εικόνα 1.3.

Η αρχιτεκτονική κοινής μνήμης έχει 2 κατηγοριοποιήσεις. Αν ο χρόνος πρόσβασης στην καθολική μνήμη είναι ίδιος για όλους τους επεξεργαστές, τότε η οργάνωση αυτή της μνήμης λέγεται Uniform Memory Access (UMA). Αν κάποιες προσβάσεις μνήμης είναι γρηγορότερες από κάποιες άλλες, ανάλογα με το ποιος επεξεργαστής ζητάει την πρόσβαση μνήμης, τότε η οργάνωση της μνήμης λέγεται Non-Uniform Memory Access (NUMA). Οι NUMA αρχιτεκτονικές έχουν χαμηλή καθυστέρηση για προσβάσεις στην κοντινή με τον επεξεργαστή μνήμη και υψηλό εύρος ζώνης μνήμης (memory bandwidth). Οι επε-



Σχήμα 1.3: Κλασσική οργάνωση μιας SMP αρχιτεκτονικής.

ξεργαστές στην αρχιτεκτονική κοινής μνήμης μπορούν να εκτελούν εργασίες παράλληλα χρησιμοποιώντας τα ίδια κοινά δεδομένα, αλλά μπορούν να συμβούν καταστάσεις συναγωνισμού. Ως αποτέλεσμα, οι επεξεργαστές χρειάζονται να συντονιστούν προκειμένουν να αποφευχθούν ταυτόχρονες προσβάσεις στα ίδια κοινά δεδομένα. Έτσι, σε αυτές τις περιπτώσεις χρησιμοποιούνται μηχανισμοί συγχρονισμού, όπως κλειδώματα, και ατομικές μεταβλητές.

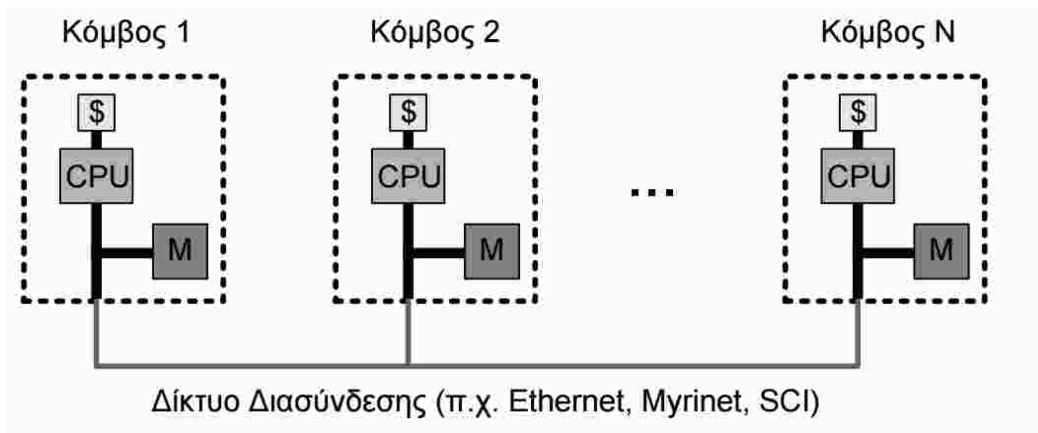
Η αποτελεσματική και εύκολη πρόσβαση στα κοινά δεδομένα από οποιοδήποτε επεξεργαστή μέσω μιας απλής εντολής load ή store καθιστά την αρχιτεκτονική κοινής μνήμης πολύ ελκυστική για τον παράλληλο προγραμματισμό. Ωστόσο, αυτή η οργάνωση της μνήμης έχει έναν κοινό δίαυλο που διασυνδέει όλους τους επεξεργαστές και περιορισμένο εύρος ζώνης μνήμης. Επομένως, αυτή η αρχιτεκτονική δε μπορεί να χρησιμοποιηθεί για πάνω από 20 ή 30 επεξεργαστές εξαιτίας του περιορισμένου εύρους ζώνης μνήμης.

1.3.2 Αρχιτεκτονική κατανεμημένης μνήμης

Στην αρχιτεκτονική κατανεμημένης μνήμης κάθε επεξεργαστής έχει μία τοπική ιεραρχία μνήμης και μία τοπική κεντρική μνήμη. Οι επεξεργαστές συνδέονται σε ένα δίκτυο διασύνδεσης (παράδειγμα Ethernet) και λέγονται κόμβοι. Δεν έχουν κοινές διευθύνσεις μνήμης και ο μόνος τρόπος για να επικοινωνούν μεταξύ τους είναι μέσω μηνυμάτων στο δίκτυο διασύνδεσης. Το σύστημα παρέχει στον προγραμματιστή ρουτίνες για αποστολή και λήψη μηνυμάτων προς και από τον επεξεργαστή. Η εικόνα 1.4 απεικονίζει την οργάνωση μνήμης σε μία αρχιτεκτονική κατανεμημένης μνήμης.

Αυτή η αρχιτεκτονική χρησιμοποιείται σε cluster τα οποία είναι συλλογές από εμπορικούς υπολογιστές που συνδέονται μεταξύ τους πάνω από ένα I/O δίκτυο διασύνδεσης. Κάθε επεξεργαστής έχει ένα ξεχωριστό αντίγραφο του λειτουργικού συστήματος.

Ένα μειονέκτημα των clusters είναι το κόστος διαχείρισης. Το κόστος διαχείρισης



Σχήμα 1.4: Κλασσική οργάνωση μνήμης σε μια αρχιτεκτονική κατανεμημένης μνήμης.

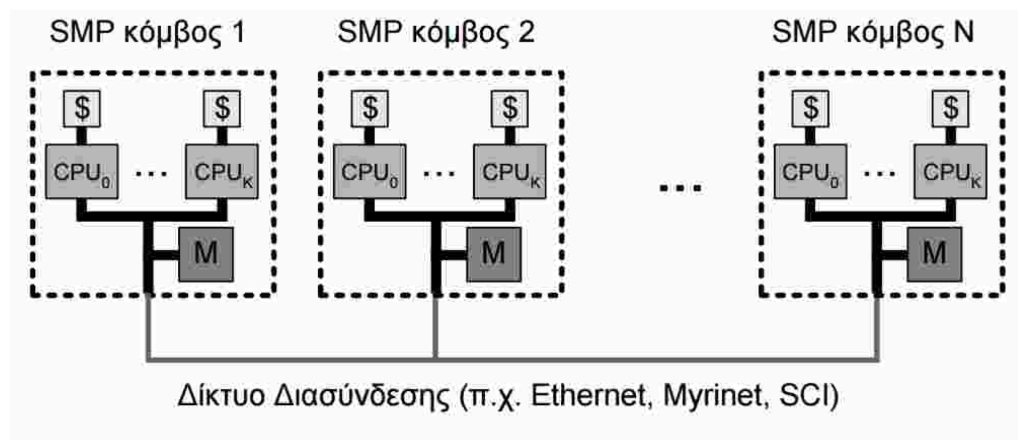
ενός cluster με n κόμβους είναι ίδιο με το κόστος διαχείρισης n υπολογιστών, ενώ το κόστος διαχείρισης ενός πολυεπεξεργαστικού συστήματος με n πυρήνες είναι ίδιο με αυτό του ενός υπολογιστή. Επίσης, ένα αλλο μειονέκτημα των cluster είναι το εύρος ζώνης. Οι επεξεργαστές σε ένα culster συνδέονται χρησιμοποιώντας I/O διασύνδεση, ενώ σε ένα πολυεπεξεργαστικό σύστημα χρησιμοποιείται διασύνδεση μνήμης.

Ο προγραμματισμός σε μια αρχιτεκτονική κατανεμημένης μνήμης αποτελεί μια πρόκληση, καθώς κάθε προσπάθεια επικοινωνίας μεταξύ των επεξεργαστών πρέπει να καθορίζεται εκ των προτέρων. Μια αποτελεσματική παραλληλοποίηση απαιτεί την πλήρη κατανόηση των εξαρτήσεων μνήμης του προγράμματος και μία αποτελεσματική κατανομή της μνήμης απο την αρχή με σκοπό να ελαχιστοποιηθεί η επικοινωνία μεταξύ απομακρυσμένων επεξεργαστών.

Τέλος, στα κατανεμημένα συστήματα μπορεί να επιτευχθεί υψηλή κλιμακωσιμότητα εξαιτίας της απουσίας κοινής μνήμης και καταστάσεων συναγωνισμού. Τα κατανεμημένα συστήματα κατασκευάζονται από χιλιάδες ανεξάρτητους κόμβους οι οποίοι μπορούν να εισαχθούν και να αφαιρεθουν δυναμικά από το δίκτυο.

1.3.3 Υβριδική αρχιτεκτονική μνήμης

Η υβριδική αρχιτεκτονική συνδυάζει τις δύο προηγούμενες αρχιτεκτονικές και επωφελείται από τα πλεονεκτήματα και των δύο. Ένα υβριδικό σύστημα είναι σαν ένα κατανεμημένο σύστημα, στο οποίο ένας συμμετρικός πολυεπεξεργαστής έχει πάρει τη θέση ενος επεξεργαστικού κόμβου. Η εικόνα Figure 1.5 παρουσιάζει μία υβριδική αρχιτεκτονική. Αυτή η τυπική αρχιτεκτονική χρησιμοποιείται σε clusters και υπερυπολογιστές και κλιμακώνει σαν μια αρχιτεκτονική κατανεμημένης μνήμης.



Σχήμα 1.5: Κλασσική οργάνωση μια Υβριδικής αρχιτεκτονικής.

1.4 Συγχρονισμός

Στον παράλληλο προγραμματισμό, χρειάζεται τα νήματα ή οι διεργασίες * να επικοινωνούν μεταξύ τους και να εκτελούν λειτουργίες στα ίδια κοινά δεδομένα ή μεταβλητές. Οι λειτουργίες ή ο κώδικας μίας διεργασίας πρέπει να εκτελούνται σαν η διεργασία να τρέχει απομονωμένα από τις υπόλοιπες. Ωστόσο, όταν διεργασίες τρέχουν παράλληλα και εκτελούν λειτουρίες σε κοινές δομές δεδομένων απαιτείται ένας μηχανισμός συγχρονισμού, αλλιώς το αποτέλεσμα των λειτουργιών θα είναι απροσδιόριστο.

Ο συγχρονισμός διεργασιών ορίζεται σαν μία τεχνική κατά την οποία πολλές ταυτόχρονες διεργασίες δε μπορούν να εκτελέσουν ταυτόχρονα ένα τμήμα του προγράμματος που καλείται **κρίσιμο τμήμα.** Το κρίσιμο τμήμα είναι ένα σειριακό τμήμα του προγράμματος. Όταν μία διεργασία αρχίζει να εκτελεί το κρίσιμο τμήμα, οι υπόλοιπες πρέπει να περιμένον μέχρι η πρώτη να ολοκληρώσει την εκτέλεση του κρίσιμου τμήματος. Αν δεν εφαρμοσθεί κάποιο είδος συγχρονισμού, οι τιμές των μεταβλητών δε μπορούν να προβλεφθούν και εξαρτώνται από τα context switches μεταξύ των διεργασιών.

Υπάρχουν διάφορες τεχνικές συγχρονισμού:

- Αμοιβαίος αποκλεισμός
  Ο αμοιβαίος αποκλεισμός είναι η πιο διαδεδομένη blocking τεχνική συγχρονισμού. Μπορεί να υλοποιηθεί μέσω μηχανισμών όπως οι σημαφόροι και τα κλειδώματα. Σχεδόν όλα τα κλειδώματα χρησιμοποιούν την Test-And-Set (TAS) ατομική εντολή. Η διεργασία που θέτει το κλείδωμα σε κατάσταση LOCKED λέμε ότι έχει πάρει το κλείδωμα και μπορεί προχωρήσει στο κρίσιμο τμήμα.

  Η εντολή TAS είναι ατομική και μόνο μία διεργασία μπορεί να θέσει τη θέση μνήμης μία δεδομένη στιγμή. Αν η θέση μνήμης είναι σε κατάσταση UNLOCKED,

*Θα χρησιμοποιήσουμε τον όρο διεργασία για να αναφερθούμε σε μία παράλληλη εργασία, αλλά στον παράλληλο προγραμματισμό μπορούμε να έχουμε τόσο διεργασίες όσο και νήματα.

τότε η διεργασία μπορεί να τη θέσει σαν LOCKED και να προχωρήσει. Διαφορετικά, αν η θέση μνήμης είναι σε κατάσταση LOCKED η διεργασία περιμένει ενώ ελέγχει και την κατάσταση της θέσης μνήμης μέχρι αυτήν να γίνει UNLOCKED.

Με την ατομική εντολή TAS μία διεργασία θέτει την κατάσταση μιας θέσης μνήμης σε LOCKED και διαβάζει την προηγούμενη τιμή της για να προχωρήσει ή να περιμένει. Αυτό όμως προκαλεί μεγάλη κίνηση στο δίαυλο εξαιτίας του πρωτοκόλλου συνάφειας. Ως αποτέλεσμα, μία βελτίωση της TAS είναι η ατομική εντολή Test-and-Test-And-Set (TTAS), η οποία πρώτα διαβάζει την κατάσταση της θέσης μνήμης και προσπαθεί να τη θέσει σε LOCKED μόνο όταν αύτη φαίνεται να έχει πάρει την τιμή UNLOCKED. Η διεργασία δεν θέτει συνεχώς τη θέση μνήμης ενώ περιμένει, αλλά τη διαβάζει τοπικά και έτσι η κίνηση λόγω συνάφειας μειώνεται.

Η υλοποίηση συγχρονισμού με κλειδώματα δεν είναι τόσο εύκολη και μερικές φορές οι διεργασίες χρειάζεται να πάρουν περισσότερα του ενός κλειδώματα. Μία προβληματική κατάσταση που προκύπτει κατά το συγχρονισμό με κλειδώματα είναι το αδιέξοδο (deadlock). Το αδιέξοδο είναι μια κατάσταση κατά την οποία δύο ή περισσότερες ανταγωνιστικές διεργασίες περιμένουν η μία κάποια άλλη να τελειώσει. Παράδειγμα αποτελεί η περίπτωση που δύο νήματα που κατέχουν δύο διαφορετικά κλειδώματα και το ένα νήμα προσπαθεί να αποκτήσει το κλείδωμα του άλλου.

- Ατομικές εντολές

  Ένα πρόβλημα με τον αμοιβαίο αποκλεισμό είναι ότι αν ένα νήμα που κρατάει το κλειδωμα αναστείλει τη λειτουργία του, τα υπόλοιπα νήματα θα μπλοκάρονται μέχρι αυτό να συνεχίσει, καθώς ο αμοιβαίος αποκλεισμός είναι blocking μηχανισμός. Για να αποφύγουμε αυτό το πρόβλημα χρησιμοποιούμε τις ατομικές εντολές. Με τις ατομικές εντολές μόνο ένας επεξεργαστής μπορεί να διαβάσει μία διεύθυνση μνήμης και να γράψει σε αυτήν. Αυτό προλαμβάνει άλλους επεξεργαστές να γράψουν ή να διαβάσουν αυτή τη θέση μνήμης τη δεδομένη χρονική στιγμή. Όταν μία διεργασία εκτελεί μία ατομική εντολή αυτό φαίνεται σαν να συμβαίνει στιγμιαία.

  Το πλεονέκτημα των ατομικών εντολών συγκριτικά με τα κλειδώματα είναι ότι αποφεύγονται τα αδιέξοδα και αποτελούν ένα non-blocking μηχανισμό. Το κύριο μειονέκτημά τους είναι ότι μπορούν να εκτελέσουν ένα περιορισμένο set εντολών. Ένα παράδειγμα ατομικής εντολής είναι η Compare And Swap (CAS). Αυτή συγκρίνει το περιεχόμενο μιας θέσης μνήμης με μία δοσμένη τιμή και μόνο όταν αυτές είναι ίδιες αλλάζει το περιεχόμενο της θέση μνήμης με μία νέα τιμή. Αν η τιμή είναι ενημέρωμένη, η εντολή είναι επιτυχής. Διαφορετικά, η τιμή έχει αλλάξει στο ενδιάμεσο από μία άλλη διεργασία και η εντολή πρέπει να επανεκκινήσει.

- Transactional Memory

  Ένα άλλο non-blocking σχήμα για συγχρονισμό είναι η transactional memory. Η transactional memory είναι μία τεχνολογία που χρησιμοποιείται για το συγχρονισμό ταυτόχρονων διεργασιών και απλοποιεί τον παράλληλο προγραμματισμό. Εκτελεί γκρουπ εντολών σαν ατομικές δοσοληψίες. Τα κύρια οφέλη αυτής της τε-

χνολογίας είναι ότι δεν υπάρχουν κλειδώματα και αδιέξοδα, το επίπεδο του παραλληλισμού είναι αυξημένο, ομοίως και η απόδοση και η παραγωγή κώδικα με αυτή την τεχνολογία είναι σχετικά απλή.

Κεφάλαιο 2

Παράλληλα δυαδικά δέντρα αναζήτησης

2.1 Παράλληλες δομές δεδομένων

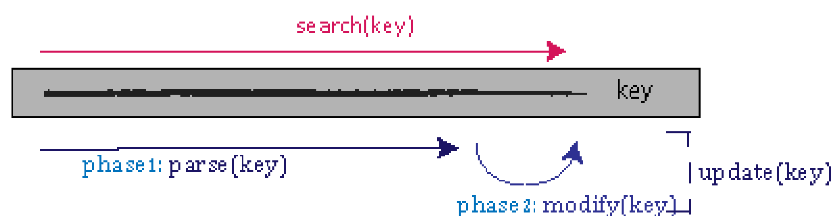
Μία δομή δεδομένων είναι ένας συγκεκριμένος τρόπος αποθήκευσης και οργάνωσης δεδομένων σε έναν υπολογιστή, έτσι ώστε αυτά να χρησιμοποιούνται αποδοτικά. Οι δομές δεδομένων είναι ένα μέσο για να διαχειριζόμαστε πολλά δεδομένα αποδοτικά και χρησιμοποιούνται σε βάσεις δεδομένων και υπηρεσίες ευρητηρίων. Διαφορετικές εφαρμογές απαιτούν διαφορετικά είδη δομών δεδομέων και η απόδοση της εφαρμογής επηρεάζεται σημαντικά από τη δομή δεδομένων που χρησιμοποιεί.

Στις μέρες μας η κυρίαρχη πλατφόρμα εκτέλεσης υπολγισμών είναι οι πολυπύρηνοι υπολογιστές. Για να τρέξουν σε αυτά τα συστήματα οι δόμες δεδομένων έπρεπε να επανασχεδιαστούν και να συγχρονίζονται οι προσβάσεις σε αυτές. Πολλαπλά νήματα μπορούν να κάνουν πρόσβασεις στη δομή δεδομένων ταυτόχρονα, αφού αυτά τρέχουν ταυτόχρονα σε διαφορετικούς πυρήνες. Ωστόσο, ο παράλληλος προγραμματισμός εισάγει διάφορες δυσκολίες. Η υλοποίηση παράλληλων δομων δεδομένων είναι αρκετά πολύπλοκη και απαιτεί συνέπεια δομής.

Ο βασικός περιορισμός στη σχεδίαση παράλληλων δομών δεδομένων είναι το σειριακό μέρος του αλγορίθμου. Είναι επιθυμητό οι δομές δεδομένων να δίνουν καλύτερη απόδοση με την αύξηση του αριθμού των νημάτων. Αυτές οι δομές ονομάζονται κλιμακώσιμες (scalable). Ο συγχρονισμός μεταξύ των νημάτων υποβαβμίζει την κλιμακωσιμότητα. Επίσης, ένα άλλο πρόβλημα του παράλληλου προγραμματισμού είναι η συμφόρηση μνήμης. Πολλαπλά νήματα απαιτούν προσβάσεις στα ίδια δεδομένα και αυτό δημιουργεί κίνηση στο δίαυλο μνήμης και περιορίζει την κλιμακωσιμότητα. Ο στόχος των προγραμματιστών που σχεδιάζουν παράλληλες δομές δεδομένων είναι να μειώσουν το σειριακό μέρος του αλγορίθμου και τα κόστη συγχρονισμού μεταξύ των νημάτων.

2.2 Δέντρα αναζήτησης

Το δέντρο αναζήτησης είναι μία δομή δεδομένων στην οποία κάθε στοιχείο της έχει ένα ζεύγος κλειδιού-τιμής και προσδιρίζει συγκεκριμένες τιμές ενός σετ. Υπάρχουν τρεις βασικές λειτουργίες σε αυτή τη δομή, η αναζήτηση, η εισαγωγή και η διαγραφή ενός στοιχείου, όπως φαίνεται στην εικόνα 2.1. Με σκοπό να μειωθεί ο χρόνος αναζήτησης ενός στοιχείου κάποια δέντρα αναζήτησης είναι ισοζυγισμένα, δηλαδή όλα τα φύλλα έχουν σχετικά ίδιο βάθος στο δέντρο. Σε αυτό το κεφάλαιο θα μελετήσουμε τρεις διαφορετικούς τύπους δέντρων αναζήτησης: τα απλά δυαδικά δέντρα αναζήτησης (Binary Search Trees BST), τα AVL δέντρα και τα Red-black δέντρα (RBT).

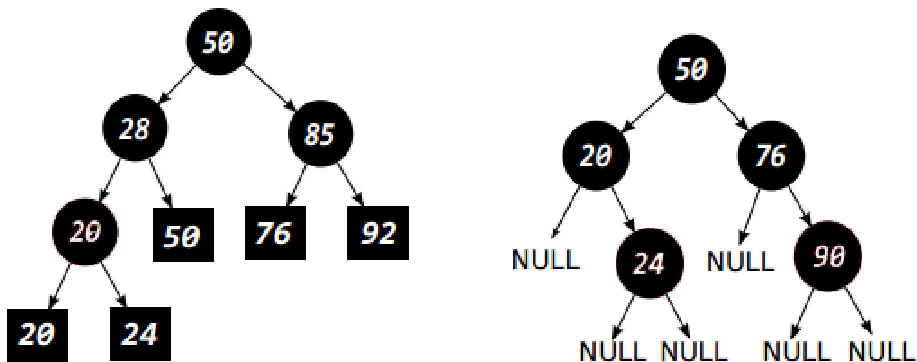


Σχήμα 2.1: Το intreface μιας δομής αναζήτησης. Οι λειτουργίες ενημέρωσης έχουν δύο φάσεις: μία φάση αναζήτησης του στοιχείου και μία φάση εκτέλεσης της ενημέρωσης στη δομή.

Τα δέντρα αναζήτησης χωρίζονται σε δύο κατηγορίες ανάλογα με την οργάνωση των ζευγών κλειδιού-τιμής. Οι κόμβοι που έχουν παιδιά λέγονται εσωτερικοί κόμβοι και αυτοί που δέν έχουν παιδιά λέγονται εξωτερικοί κόμβοι. Όταν τα ζεύγη κλειδιού-τιμής αποθηκεύονται μόνο σε εξωτερικούς κόμβους (φύλλα) και οι εσωτερικοί κόμβοι χρησιμοποιούνται μόνο ως κόμβοι δρομολόγησης τότε το δέντρο λέγεται external. Όταν οι εσωτερικοί κόμβοι δεν είναι μόνο κόμβοι δρομολόγησης αλλά αποθηκεύουν και ζεύγη κλειδιού-τιμής, τα δέντρα αναζήτησης λέγονται internal. Η εικόνα 2.2 απεικονίζει ένα external και ένα internal δέντρο αναζήτησης. Τα τετράγωνα σχήματα χρησιμοποιούνται για να διαχωρίσουν τα φύλλα που περιέχουν ζεύγη κλειδιού-τιμής από τους εσωτερικούς κόμβους δρομολόγησης στο external δέντρο.

2.2.1 Απλά δυαδικά δέντρα αναζήτησης (BST)

Ένα δυαδικό δέντρο αναζήτησης είναι ένα ταξινομημένο δέντρο στο οποίο κάθε κόμβος έχει μέχρι δύο παιδιά-κόμβους. Αυτό το δέντρο δεν είναι ισοζυγισμένο και κάθε κόμβος-παιδί είναι είτε φύλλο είτε ρίζα ενός άλλου δυαδικού δέντρου αναζήτησης. Κάθε εσωτερικός κόμβος έχει ένα κλειδί και έχει δύο υπόδεντρα, το αριστερό και το δεξί υπόδεντρο. Το δέντρο ικανοποιεί τη δυαδική ιδιότητα αναζήτησης, η οποία αναφέρει ότι το κλειδί κάθε κόμβου πρέπει να είναι μεγαλύτερο από όλα τα κλειδιά του αριστερού



Σχήμα 2.2: Ένα παράδειγμα ενός external και ενός internal δέντρου.

υπόδεντρου και μικρότερο από όλα τα κλειδιά του δεξιού υπόδεντρου. Δεν επιτρέπονται κόμβοι που να έχουν το ίδιο κλειδί.

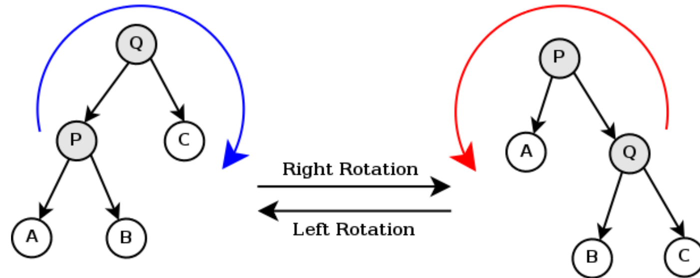
Η βασική ιδέα είναι τα κλειδιά του δέντρου να είναι σε ταξινομημένη σειρά, για να μπορούν να εκτελεστούν αποδοτικά οι λειτουργίες στο δέντρο. Κατά μέσο όρο, κάθε λειτουργία παραλείπει τος μισούς κόμβους του δέντρου, έτσι ώστε να παίρνει χρόνο ανάλογο με το λογάριθμο του αριθμού των κόμβων n του δέντρου ($\mathcal{O}(\log n)$). Ωστόσο, στη χειρότερη περίπτωση το δέντρο εκφυλλίζεται σε γραμμική αλυσίδα κόμβων (λίστα) και η λειτουργία διαρκεί χρόνο $\mathcal{O}(n)$.

2.2.2 AVL Δέντρα

Το AVL δέντρο είναι ένα ισοζυγισμένο δέντρο με βάση το ύψος του που ονομάστηκε έτσι από τους εφερεύτες του Georgy Adelson-Velsky και Evgenii Landis [3]. Ήταν το πρώτο δυναμικά ισοζυγισμένο δέντρο που προτάθηκε και έχει δύο ιδιότητες. Η πρώτη ιδιότητα είναι ότι κάθε υπόδεντρο είναι ομοίως ένα AVL δέντρο. Υποθέτοντας ότι το ύψος του δέντρου είναι ο αριθμός των κόμβων στο μακρύτερο μονοπάτι από τη ρίζα σε ένα φύλλο, η δεύτερη ιδιότητα αναφέρει ότι τα ύψη δύο υπόδεντρων-παιδιών πρέπει να διαφέρουν το πολύ κατά ένα. Αυτή η διαφορά λέγεται παράγοντας ισοζυγισμού (**balance factor**). Αν οποιαδήποτε στιγμή ο παράγοντας ισοζυγισμού είναι μεγαλύτερος του ενός, απαιτείται εξισορρόπηση (rebalancing) στο δέντρο. Αυτό επιτυγχάνεται εκτελώντας μία ή περισσότερες περιστροφές στο δέντρο. Όλες οι λειτουργίες ενός AVL δέντρου παίρνουν χρόνο ανάλογο του λογαρίθμου του αριθμού των κόμβων του δέντρου ($\mathcal{O}(\log n)$) και στη χειρότερη περίπτωση.

Όταν εκτελείται μία λειτουργία ενημέρωσης (insertion, deletion) στο δέντρο, μπορεί να χρειαστεί rebalancing, το οποίο μπορεί να ελεγχθεί με τη βοήθεια του παράγοντα ισοζυγισμού. Όταν αυτός ο παράγοντας ενός κόμβου είναι μικρότερος του -1 ή μεγαλύτερος του +1, το υπόδεντρου με ρίζα το συγκεκριμένο κόμβο δεν είναι ισοζυγισμένο. Έτσι, πρέπει να εκτελεστουν αριστερές και δεξιές περιστροφές στο δέντρο για να ισοζυγιστεί

2.3.



Σχήμα 2.3: Μία δεξιά και μία αριστερή περιστροφή δέντρου.

2.2.3 Red-Black Δέντρα

Ένα Red-Black δέντρο είναι επίσης ένα δέντρο ισοζυγισμένο στο ύψος του. Βασίζεται στα συμμετρικά δυαδικά B-tree και περιγράφεται στη δημοσίευση με τίτλο "A Dichromatic Framework for Balanced Trees" [4]. Κάθε κόμβος ενός Red-Black δέντρου έχει ένα επιπλέον bit το οποίο αντιπροσωπεύει το χρώμα του κόμβου και μπορεί να είναι κόκκινο ή μαύρο. Αυτό το χρώμα χρησιμοποιείται για να διατηρήσει την ισορροπία του δέντρου. Τα RBT δέντρα ικανοποιούν κάποιες συγκεκριμένες ιδιότητες που ονομάζονται coloring properties και βεβαιώνουν ότι το δέντρο είναι σχεδόν ισορροπημένο. Όταν εκτελούνται λειτουργίες ενημέρωσης όπως εισαγωγή ή διαγραφή το δέντρο μπορεί να χρειαστεί να αναδιαταχθεί και κάποιοι κόμβοι του να αλλάξουν χρώμα για να διατηρηθούν οι coloring properties.

Αυτές είναι οι ακόλουθες:

1. Ένας κόμβος είναι είτε κόκκινος είτε μαύρος.

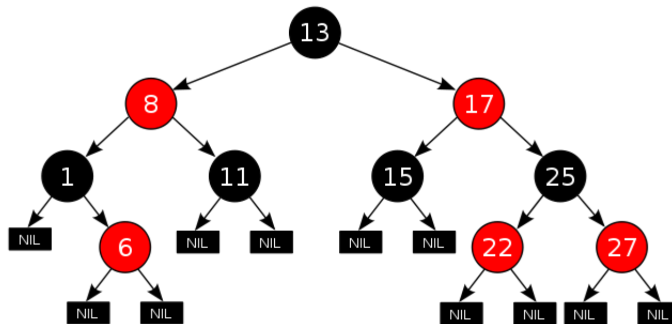
2. Η ρίζα του δέντρου είναι μαύρη (root property).

3. Κάθε φύλλο είναι μαύρο.

4. Αν ένας κόμβος είναι κόκκινος, τότε και τα δύο παιδιά του πρέπει να είναι μαύρα (red property).

5. Κάθε μονοπάτι από έναν κόμβο σε οποιοδήποτε φύλλο έχει το ίδιο πλήθος μαύρων κόμβων (black property). Ο αριθμός των μαύρων κόμβων από τη ρίζα σε ένα κόμβο είναι το βάθος του κόμβου και ο ίδιος αριθμός μαύρων κόμβων σε όλα τα μονοπάτια προς τα φύλλα λέγεται black-height του Red–Black δέντρου.

Αυτές οι ιδιότητες έχουν σχεδιασθεί με τέτοιο τρόπο έτσι ώστε το δέντρο να αναδιατάσσεται αποδοτικά. Η εικόνα 2.4 δείχνει ένα παράδειγμα ενός Red-Black δέντρου.



Σχήμα 2.4: Ένα παράδειγμα ενός red–black δέντρου.

Σύμφωνα με τις ιδιότητες το μεγαλύτερο μονοπάτι σε ένα δέντρο δε μπορεί να είναι μακρύτερο από το διπλάσιο του συντομότερου μονοπατιού, αφού όλα τα μονοπάτια έχουν τον ίδιο αριθμό μαύρων κόμβων.

Οι βασικές λειτουργίες (lookup, insertion, deletion) απαιτούν στη χειρότερη περίπτωση χρόνο ανάλογο του ύψους του δέντρου $\mathcal{O}(\log n)$, όπου n είναι ο συνολικός αριθμός των κόμβων στο δέντρο. Αυτό το θεωρητικό άνω όριο ύψους επιτρέπει στο Red-Black δέντρο να είναι αποδοτικό και στη χειρότερη περίπτωση. Όταν εισάγεται ένας κόμβος ή διαγράφεται ένας κόμβος συνήθως παραβιάζονται οι παραπάνω ιδιότητες και το δέντρο χάνει την ισορροπία του. Υπάρχουν δύο πιθανές παραβιάσεις, η *red-red violation*, όταν ένας κόκκινος κόμβος αποκτήσει ένα κόκκινο παιδί και παραβιάζεται η red property και η *double black violation*, όταν ένα μονοπάτι του δέντρου περιέχει λιγότερους μαύρους κόμβους από άλλα μονοπάτια και παραβιάζεται η black property. Για να αντιμετωπιστούν αυτές οι παραβιάσεις εκτελούνται περιστροφές και αλλαγές χρωματισμών στους κόμβους του δέντρου.

2.3 Τεχνικές κατασκευής παράλληλων δομών δεδομένων

Στις παράλληλες δυαδικές δομές μπορούν να εκτελούνται ταυτόχρονα λειτουργίες που αλλάζουν τη δομή. Προκειμένου να βεβαιωθούμε ότι παράγονται σωστά αποτελέσματα και η δομή παραμένει συνεπής απαιτείται ένας μηχανισμός συγχρονισμού στη δομή. Στις βασικές λειτουργίες της δομής χρησιμοποιούνται διάφορες τεχνικές συγχρονισμού όπως ο αμοιβαίος αποκλεισμός και οι ατομικές λειτουργίες. Οι τρεις πιο διαδεδομένες τεχνικές συγχρονισμού για παράλληλα δέντρα αναζήτησης είναι: coarse-grained locking, fine-grained locking και lock-free programming.

2.3.1 Coarse-grained locking

Coarse-grained locking είναι μία τεχνική κατασκευής παράλληλων δυαδικών δομών χρησιμοποιώντας τον αμοιβαίο αποκλεισμό και τα κλειδώματα. Ως lock granularity ορίζεται ένα μέτρο του ποσού των δεδομένων που ένα κλείδωμα προστατεύει. Στη coarse-grained locking τεχνική όταν μία διεργασία/νήμα χρειάζεται να κάνει πρόσβαση σε κοινά δεδομένα, όλες οι προσβάσεις στα κοινά δεδομένα προστατεύονται με ένα καθολικό κλείδωμα (global lock). Εκτελούνται οι απαραίτητες read/write λειτουργίες και μετά το κλείδωμα απελευθερώνεται. Οι προσβάσεις στα κοινά δεδομένα σειριοποιούνται και μόνο μία διεργασία/νήμα μπορεί να έχει πρόσβαση σε αυτά. Οπότε ουσιαστικά δεν υπάρχει παραλληλισμός. Αυτή η τεχνική είναι εύκολη στην υλοποίηση και στη χρήση. Το μειονέκτημά της είναι ότι περιορίζει την απόδοση ενός πολυπεπεξεργαστικού συστήματος. Στην περίπτωση πολλαπλών νημάτων αυτά χρειάζεται να περιμένουν μέχρι το νήμα που κρατάει το καθολικό κλείδωμα να τελειώσει τη λειτουργία του και να απελευθερωσει το κλείδωμα. Αυτή η περίπτωση δημιουργεί υψηλή συμφόρηση στο κλείδωμα και υποβαθμίζει την απόδοση. Η coarse-grained locking τεχνική είναι χρήσιμη όταν το κρίσιμο τμήμα είναι μικρό και δε δημιουργείται υψηλή συμφόρηση στην απόκτηση του κλειδώματος.

2.3.2 Fine-grained locking

Σε αυτή την τεχνική χρησιμοποιούνται πολλαπλά κλειδώματα που προστατεύουν μικρό μέρος της δομής. Αυτό προκαλεί αυξημένο lock overhead, γιατί χρησιμοποιούνται πολλά κλειδώματα, απαιτείται περισσότερη μνήμη και υπάρχει επιπλέον κόστος λόγω συχνων λειτουργιών απόκτησης/απελευθέρωσης των κλειδωμάτων. Από την άλλη πλευρά, το fine-grained locking επιτρέπει υψηλό παραλληλισμό γιατί μειώνει τη συμφόρηση στην απόκτηση κλειδωμάτων και έχει καλή κλιμακωσιμότητα. Πολλαπλές διεργασίες/νήματα μπορούν να προχωρήσουν παράλληλα όταν δεν κάνουν πρόσβαση στα ίδια μέρη της κοινής δομής δεδομένων. Ωστόσο, αυτή η τεχνική είναι αρκετά σύνθετη και δύσκολη στην υλοποίση, αφού απαιτείται ο προγραμματιστής να γνωρίζει εκ των προτέρων ποια κλειδώματα θα αποκτηθούν από το κάθε νήμα και με ποια σειρά. Τα νήματα πρέπει να καταλαμβάνουν τα κλειδώματα με την ίδια κατευθυνση (global order) για να αποφευχθούν τα αδιέξοδα (deadlocks).

2.3.3 Lock-free προγραμματισμός

Ο lock-free προγραμματισμός δε χρησιμοποιεί κλειδώματα και βελτιώνει την απόδοση του συστήματος και την ευρωστία, αφού είναι ένας non-blocking μηχανισμός. Οι δομές που χρησιμοποιούν αυτήν την τεχνική λέγονται non-blocking και μπορεί να είναι wait-free, όταν κάθε λειτουργία εγγυάται ότι θα τελειώσει σε ένα ορισμένο αριθμό βηματων, lock-free όταν κάποιες λειτουργίες μπορούν να τελειώσουν σε ορισμένο αριθμό βημάτων και obstruction-free, όταν μία λειτουργία μπορεί να τελειώσει σε ορισμένο αριθμό βημάτων. Το κλειδί στο lock-free προγραμματισμό είναι η υποστήριξη από το υλικό. Οι non-blocking δομές χρησιμοποιούν τις ατομικές εντολές. Αυτές εγγυώνται ατομικότητα, δηλαδή ότι τα νήματα μπορουν να δουν την κατάσταση πριν ή μετά από μία λει-

τουργία και όχι μία ενδιάμεση κατάσταση. Οι πιο διαδεδομένες ατομικές εντολές είναι η compare-and-swap (CAS), test-and-set (TAS), test-and-test-and-set (TTAS) και η load-linked/store-conditional (LL/SC) εντολή.

2.4 Βασικές λειτουργές παράλληλων δέντρων αναζήτησης

Όπως έχει ήδη αναφερθεί, ένα παράλληλο δέντρο αναζήτησης είναι ένα σύνολο στοιχείων. Οι βασικές λειτουργίες που μπορούν να εκτελεστούν σε αυτό είναι η αναζήτηση (lookup), η εισαγωγή (insertion) και η διαγραφή (deletion). Κάθε στοιχείο του συνόλου αυτού αποθηκεύεται σε έναν κόμβο του δέντρου και αποτελεί ένα ζεύγος κλειδιού-τιμής. Το κλειδί προσδιορίζει μοναδικά ένα στοιχείο του συνόλου. Οι βασικές λειτουργίες έχουν την ακόλουθη σημασιολογία:

- **lookup(key)**: αναζητεί έναν κόμβο με το δοσμένο κλειδί. Αν αυτός βρεθεί, η λειτουργία επιστρέφει την τιμή που σχετίζεται με το δοσμένο κλειδί διαφορετικά επιστρέφει NULL.
- **insertion(key, value)**: προσπαθεί να εισάγει έναν καινούργιο κόμβο στο δέντρο αναζήτησης με το συγκεκριμένο δοσμένο ζεύγος κλειδιου-τιμής. Η εισαγωγή είναι επιτυχής όταν δεν υπάρχει άλλος κόμβος στο δέντρο με το ίδιο κλειδί.
- **deletion(key)**: προσπαθεί να διαγράψει από το δέντρο τον κόμβο που περιέχει το συγκεκριμένο κλειδί. Η λειτουργία είναι επιτυχής όταν υπάρχει ένας τέτοιος κόμβος στο δέντρο.

Οι δύο τελευταίες λειτουργίες (insertion και deletion) αποτελούνται από δύο διακριτές φάσεις. Πρώτα, υπάρχει μία διάσχιση στο δέντρο μέχρι να βρεθεί ο επιθυμητός κόμβος και μετά πραγματοποιείται η τροποποίηση της δομής του δέντρου στον κόμβο αυτόν.

2.5 Μία απλοϊκή προσέγγιση

Σε αυτήν την ενότητα θα παρουσιάσουμε κάποιες υλοποιήσεις απλοϊκών παράλληλων δέντρων αναζήτησης. Κάποια από αυτά υλοποιήθηκαν στις δημοσιεύσεις [5], [6]. Πριν παρουσιάσουμε τις διάφορες υλοποιήσεις θα εισάγουμε δύο προσεγγίσεις για την εξισορρόπηση ενός δέντρου μετά από μία λειτουργία ενημέρωσης που παραβιάζει τις ιδιότητες του. Αυτές οι προσεγγίσεις σχετίζονται με τη σειρά με την οποία εκτελείται η εξισορρόπηση του δέντρου και η λειτουργία ενημέρωσης. Η εισαγωγή και η διαγραφή σε ένα δέντρο έχουν δύο φάσεις. Στην πρώτη φάση υπάρχει μία διάσχιση του δέντρου με ένα top-down τρόπο, π.χ από τη ρίζα προς τα φύλλα μέχρι να βρεθεί ο κατάλληλος κόμβος ανάλογα με τη λειτουργία που εκτελείται. Έπειτα, εκτελείται η λειτουργία ενημέρωσης και υπάρχει μία διάσχιση του δέντρου με έναν bottom-up τρόπο με κατεύθυνση προς τη

ρίζα του δέντρου κατά την οποία εκτελούνται οι κατάλληλες τροποποιήσεις στη δομή του δέντρου (π.χ. περιστροφές) για την εξισορρόπησή του. Αυτή η προσέγγιση λέγεται bottom-up. Όταν οι λειτουργίες ενημέρωσης (insertion, deletion) εκτελούνται με μία μόνο top-down διάσχιση του δέντρου η προσέγγιση καλείται top-down. Προκειμένου να είναι ισοζυγισμένο το δέντρο στην top-down υλοποίηση εκτελούνται περιστροφές και κατάλληλες τροποποιήσεις στη δομή του δέντρου εκ των προτέρων κατά την top-down διάσχιση και όταν φθάσουμε στον επιθυμητό κόμβο για να γίνουν οι αλλάγες, πραγματοποιείται η ενημέρωση και η δομή είναι συνεπής.

2.5.1 Περιγραφή

Έχουμε αναπτύξει εννέα διαφορετικές υλοποιήσεις παράλληλων δέντρων αναζήτησης, τόσο internal και externa. χρησιμοποιώντας τις τεχνικές συγρονισμού που αναφέρθηκαν και υπάρχουν τόσο bottom-up όσο και top-down προσεγγίσεις για την εξισσορόπηση του δέντρου.

Τα παράλληλα δυαδικά δέντρα αναζήτησης είναι:

- avl-bu-cg-ext-lock tree
  - AVL δέντρο
  - external δέντρο
  - coarse-grained locking
  - bottom-up προσέγγιση για την εξισορρόπηση του δέντρου
  - επαναληπτική (iterative) υλοποίηση
- bst-td-fg-int-lock tree
  - BST δέντρο
  - internal δέντρο
  - fine-grained locking
  - top-down (δε χρειάζεται εξισορρόπηση)
- rbt-bu-cg-ext-iter-lock tree
  - Red-Black δέντρο
  - external δέντρο
  - coarse-grained locking
  - bottom-up προσέγγιση για την εξισορρόπηση
  - επαναληπτική (iterative) υλοποίηση
- rbt-bu-cg-int-iter-lock tree
  - Red-Black δέντρο
  - internal δέντρο
  - coarse-grained locking
  - bottom-up προσέγγιση για την εξισορρόπηση
  - επαναληπτική (iterative) υλοποίηση

- rbt-bu-cg-ext-rec-lock tree
  - Red-Black δέντρο
  - external δέντρο
  - coarse-grained locking
  - bottom-up προσέγγιση για την εξισορρόπηση
  - αναδρομική (iterative) υλοποίηση
- rbt-td-cg-ext-lock tree
  - Red-Black δέντρο
  - external δέντρο
  - coarse grained locking
  - top-down προσέγγιση για την εξισορρόπηση
- rbt-td-fg-ext-lock tree
  - Red-Black δέντρο
  - external δέντρο
  - fine-grained locking
  - top-down προσέγγιση για την εξισορρόπηση
- rbt-td-cg-int-lock tree
  - Red-Black δέντρο
  - internal δέντρο
  - coarse grained locking
  - top-down προσέγγιση για την εξισορρόπηση
- rbt-td-fg-int-lock tree
  - Red-Black δέντρο
  - internal δέντρο
  - fine-grained locking
  - top-down προσέγγιση για την εξισορρόπηση

2.5.2 Λεπτομέρειες υλοποίησης

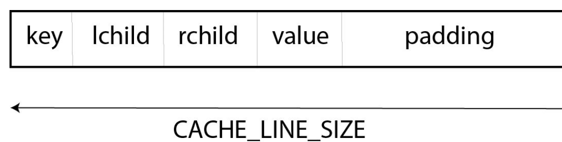
- Οι υλοποιήσεις αναπτύχθηκαν σε C γλώσσα προγραμματισμού.
- Το false sharing είναι ο πιο περιοριστικός παράγοντας κλιμακωσιμότητας στην παράλληλη εκτέλεση νημάτων στα πολυεπεξεργαστικά συστήματα. Εμφανίζεται όταν δύο ή περισσότερα νήματα προσπαθούν να τροποποιήσουν ανεξάρτητους κόμβους που βρίσκονται στην ίδια cache line. Αυτή η περίπτωση προκαλεί cache misses και υποβαθμίζει την απόδοση. Για να αποφύγουμε αυτό το πρόβλημα προσπαθούμε να ευθυγραμμίσουμε τη δομή του κόμβου στη μνήμη και χρησιμοποιούμε την τεχνική padding. Με αυτή την τεχνική ένα ή περισσότερα byte προστίθενται στη δομή του κόμβου, ετσι ώστε κάθε κόμβος να βρίσκεται σε διαφορετική cache

line. Ο κώδικας 2.1 και το σχήμα 2.5 παρουσιάζουν μία τυπική δομή bst κόμβου και μία αναπαράσταση αυτού στη μνήμη.

Listing 2.1: Μία τυπική δομή bst κόμβου

```
typedef struct bst_node{
        int key;
        struct bst_node *lchild;
        struct bst_node * rchild;
        void *value;

        char padding[CACHE_LINE_SIZE - sizeof(int) -
                     2 * sizeof(struct bst_node *) -
                     sizeof(void *)];
} bst_node_t;
```

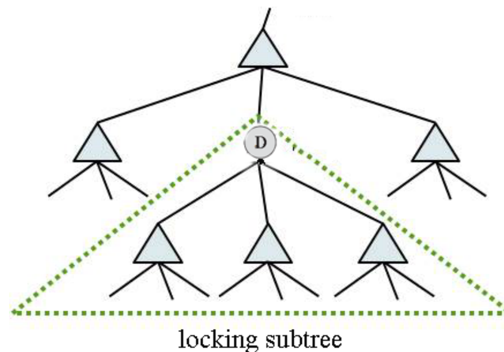


Σχήμα 2.5: Μία αναπαράσταση ενός bst κόμβου στη μνήμη χρησιμοποιώντας padding, έτσι ώστε αυτός να καταλαμβάνει ακριβώς μία cache line.

- Στην coarse-grained υλοποίηση υπάρχει ένα καθολικό κλείδωμα μοιραζόμενο για όλα τα νήματα. Κάθε νήμα που προσπαθεί να εκτελέσει μία από τις τρεις βασικές λειτουργίες περιμένει μέχρι να απελευθερωθεί το κλείδωμα, το καταλαμβάνει, εκτελεί τη λειτουργία του στη δομή και έπειτα το απελευθερώνει. Μόνο ένα νήμα μπορεί να έχει το κοινό καθολικό κλείδωμα μία δεδομένη χρονική στιγμή. Για αυτό η coarse-grained υλοποίηση είναι σειριακή υλοποίηση.

- Στη fine-grained εκδοχή των παράλληλων δέντρων αναζήτησης κάθε κόμβος έχει το δικό του κλείδωμα. Έτσι, υπάρχει ένα κλείδωμα στη δομή του κόμβου. Μία bottom-up υλοποίηση είναι δύσκολο να υλοποιηθεί γιατί δε διατηρείται η καθολική σειρά στην απόκτηση κλειδωμάτων και μπορούμε να οδηγηθούμε σε αδιέξοδο. Για αυτό το λόγο οι fine-grained υλοποιήσεις είναι όλες top-down υλοποιήσεις. Ωστόσο, κατά τη λειτουργία διαγραφής σε ένα internal δέντρο ο κόμβος της διαγραφής μπορεί να είναι ένας εσωτερικός κόμβος και ο διάδοχός του είναι ένα φύλλο, το οποίο πρέπει να πάρει τη θέση του κόμβου προς διαγραφή στο δέντρο. Αυτή η διαδικασία απαιτεί αποκλειστική πρόσβαση σε κάθε κόμβο μεταξύ αυτων των δύο. Προκειμένου να επιτευχθεί αυτό πρέπει το υπόδεντρο με ρίζα τον κόμβο προς διαγραφή να είναι κλειδωμένο και έτσι κρατάμε τον εσωτερικό κόμβο που πρόκειται

να διαγραφεί κλειδωμένο μέχρι να βρεθεί ο διάδοχος του. Η εικόνα 2.6 απεικονίζει αυτήν την περίπτωση. Αν δεν κρατήσουμε κλειδωμένο τον εσωτερικό κόμβο προς διαγραφή και τον απελευθερώσουμε και τον ξανακλειδώσουμε, αφού βρούμε το διαδοχό του, τότε αποκτούμε κλειδώματα με την αντίθετη κατεύθυνση από την καθολική σειρά (global order), κάτι που μπορεί να οδηγήσει σε αδιέξοδο. Από την άλλη, τα external δέντρα εκτελούν αλλαγές μόνο σε φύλλα και δεν εμφανίζεται αυτό το πρόβλημα.



Σχήμα 2.6: Κατά την διαγραφή ενός κόμβου D σε ένα internal δέντρο, πρέπει το υπόδεντρο με ρίζα τον κόμβο D να παραμένει κλειδωμένο.

- Στην avl υλοποίηση έχουμε προσθέσει ένα επιπλέον πεδίο στη δομή του κόμβου που αναπαριστά το ύψος του κόμβου, για να υπολογίζεται ο παράγοντας ισορροπίας και ομοίως στα Red-Black δέντρα ένα πεδίο για το χρώμα του κόμβου για να αποκαθιστούμε τις coloring properties.

2.6 Μία πιο πολύπλοκη προσέγγιση

Αυτή η ενότητα περιλαμβάνει πιο πολύπλοκες προσεγγίσεις παράλληλων δέντρων αναζήτησης. Αυτές οι υλοποιήσεις είναι σύνθετοι αλγόριθμοι τόσο lock-based όσο και lock-free που παρουσιάζονται στο [8] και χρησιμοποιούν πιο πολύπλοκους μηχανισμούς συγρονισμού. Σε αυτές τις υλοποιήσεις περιμένουμε υψηλότερη απόδοση και καλύτερη κλιμακωσιμότητα. Θα περιγράψουμε περιληπτικά την ιδέα αυτών των υλοποιήσεων.

2.6.1 Περιγραφή

Bronson

Αυτή η υλοποίηση είναι ένα relaxed ισοζυγισμένο AVL δέντρο που προτάθηκε στο [9], ελέγχει το μέγεθος του κρίσιμου τμήματος (όλες οι λειτουργίες ενημέρωσεις έχουν

ένα σταθερό κρίσιμο τμήμα) και εκμεταλλεύεται τη λογική του validation. Μία λειτουργία αναζήτησης μπορεί να μπλοκάρει μέχρι μία ταυτόχρονη λειτουργία ενημέρωσης να ολοκληρωθεί. Σε περίπτωση ταυτόχρονων λειτουργιών ενημέρωσης ο αλγόριθμος χρησιμοποιεί version numbers. Τα version numbers δείχνουν αν εκτελείται ταυτόχρονα μία λειτουργία εγγραφής. Κάθε κόμβος έχει ένα version number, για να ελέγχεται αν μία ανάγνωση είναι ακόμα έγκυρη.

Το δέντρο αυτό αναφέρεται σαν μερικώς external tree. Στα internal δέντρα η διαγραφή ενός κόμβου με δύο παιδιά απαιτεί να βρεθεί ο διάδοχος αυτού του κόμβου για να τον αντικαταστήσει στο δέντρο. Αυτός ο αλγόριθμος αφήνει και δε διαγράφει τους εσωτερικούς κόμβους δρομολογησης όταν αυτός έχει δύο παιδιά. Όταν πραγματοποιείται εξισορρόπηση στο δέντρο, οι μη έγκυροι κόμβοι δρομολόγησης που έχουν λιγότερα από δύο παιδιά διαγράφονται από το δέντρο. Η διαγραφή ενός κόμβου με λιγότερα από δύο παιδιά πραγματοποιείται άμεσα. Σε αυτή την υλοποίηση υπάρχει η ίδια δομή κόμβων για τα φύλλα και τους κόμβους δρομολόγησης, έτσι ώστε ένας κόμβος να μετατρέπεται από τη μία μορφή στην άλλη και το ανάποδο αλλάζοντας μόνο ένα πεδίο στη δομή του.

Drachsler

Αυτό το δέντρο παρουσιάζεται στο [10] και είναι ένα BST internal δέντρο που εφαρμόζει λογική διάταξη μεταξύ των κόμβων του. Το κλειδί για τη σχεδίαση αποτελεσματικών δέντρων αναζήτησης που κλιμακώνουν είναι η λειτουργία αναζήτησης να είναι αρκετά γρήγορη. Σε αυτήν την υλοποίηση η λογική διάταξη του δέντρου χωρίζεται από την φυσική διάταξη και οι λειτουργίες αναζήτησης μπορούν να προχωρήσουν ταυτόχρονα με λειτουργίες που τροποποιούν τη φυσική διάταξη του δέντρου χωρις συγχρονισμό.

Η λογική διάταξη μεταξύ των στοιχείων αναπαριστάται σαν διαδοχικά διαστήματα. Για παράδειγμα, η λογική διάταξη των στοιχείων $1 < 3 < 5 < 7 < 9$ είναι τα διαδοχικά διαστήματα $(-\infty, 1), (1, 3), (3, 5), (5, 7), (7, 9), (9, +\infty)$. Ένα στοιχείο ανήκει στο δέντρο αν και μόνο αν είναι το άκρο ενός διαστήματος, διαφορετικά δεν ανήκει στο δέντρο. Τα διαστήματα χρησιμοποιούνται για να απαντήσουν σε λειτουργίες αναζήτησης γρήγορα. Κάθε κόμβος περιέχει ένα δείκτη στο επόμενο στη λογική διάταξη κόμβο και ένα δείκτη στον προηγούμενο στη λογικη διάταξη κόμβο. Ωστόσο, απαιτείται συγχρόνισμός και σε αυτούς τους δείκτες, για αυτό υπάρχουν δύο κλειδώματα. Ένα treeLock που προστατεύει τη φυσική διάταξη του δέντρου και ένα succLock που προστατεύει τη λογική διάταξη του δέντρου. Για ένα κόμβο n, το succLock του κόμβου n προστατεύει το διάστημα $(n, succ(n))$.

BST Ticket

Αυτή η υλοποιήση προτάθηκε στο [8] και είναι ένα lock-based BST δέντρο. Προσπαθεί να μειώσει τον αριθμό των κλειδωμάτων που απαιτούνται ανά λειτουργία ενημέρωσης και κατά συνέπεια τον πλήθος των cache lines που μεταφέρονται. Πιο συγκεκριμένα, πρόκειται για ένα external δέντρο που χρησιμοποιεί version numbers στους κόμβους. Αυτοί χρησιμοποιούνται για να γίνεται αρχικά μία αισιόδοξη διάσχιση στο δέντρο και αργότερα να ανιχνευτεί πιθανό conflict λόγω παράλληλων λειτουργιών ενημέρωσης στον κόμβο.

Η απόκτηση του κλειδώματος και η επικυροποιήση του κόμβου πραγματοποιούνται σε ένα μόνο βήμα. Στις λειτουργίες ενημέρωσης υπάρχει μία διάσχιση μέχρι να βρεθεί ο κατάλληλος κόμβος, έπειτα αποκτιούνται τα κατάλληλα κλειδώματα και εκτελείται η ενημέρωση. Αν η απόκτηση κλειδωμάτων αποτύχει, σημάινει ότι το version number του κόμβου έχει αυξηθεί από μία άλλη ταυτόχρονη ενημέρωση και η λειτουργία πρέπει να ξαναρχίσει.

Aravind

Το συγκεκριμένο external δέντρο [11] είναι ένας lock-free αλγόριθμος που χρησιμοποιεί δύο ατομικές εντολές για read και write, την compare-and-swap (CAS) και την bit-test-and-set (BST). Με σκοπό να περιοριστούν οι συγκρούσεις μεταξύ των λειτουργιών ενημέρωσης υπάρχουν κάποιες βελτιστοποιήσεις. Ο συγκεκριμένος αλγόριθμος σημειώνει τις ακμές σαν διαγραμμένες και όχι τους κόμβους, δε χρησιμοποιεί επιπλέον αντικείμενα για το συντονισμό των παράλληλων λειτουργιών και επιτρέπει τη διαγραφή πολλαπλών κλειδιών σε ένα βήμα.

Σε αντίθεση με τις προηγούμενες υλοποιήσεις αυτός ο αλγόριθμος σημειώνει τις ακμές ως διαγραμμένες. Κάθε λειτουργία ενημέρωσης γίνεται ο "ιδιοκτήτης" κάποιων ακμών στις οποίες πρέπει να δουλέψει. Κάθε ακμή συνδέει δύο κόμβους και σημειώνοντας μία ακμή ως διαγραμμένη μπορεί είτε να έχουν διαγραφεί και οι δύο κόμβοι που συνδέει είτε ο κόμβος στον οποίο καταλήγει. Για να διαχωριστούν αυτές οι δύο περιπτώσεις, όταν και οι δύο κόμβοι έχουν διαγραφεί η ακμή σημειώνεται σαν *flagged*, ενώ όταν έχει διαγραφεί μόνο ο κομβος στον οποίον καταλήγει η ακμή, τότε η ακμή σημειώνεται σαν *tagged*.

Τέλος, ο αλγόριθμος χρησιμοποιεί και μία βοηθητική στρατηγική που εκτελείται μόνο στις λειτουργίες διαγραφής. Δεν υπάρχει βοηθητική στρατηγική για τις λειτουργίες εισαγωγής γιατί εισάγεται επιπλέον overhead. Η βοηθητική στρατηγική εκτελείται σε μία λειτουργία εισαγωγής όταν ανακαλυφθεί ότι η ακμή από n→parent στο n→leaf κόμβο έχει σημειωθεί σαν flagged ή tagged. Αυτό σημαίνει ότι υπάρχει μία ταυτοχρονη λειτουργία διαγραφής που προσπαθεί να διαγράψει τον κόμβο n→parent από το δέντρο. Ως αποτέλεσμα η λειτουργία εισαγωγής βοηθάει την ταυτόχρονη διαγραφή να ολοκληρωθεί και επανεκκινεί απο την αρχή. Ομοίως, μία λειτουργία διαγραφής ενος κόμβου μπορεί να ανιχνεύσει ότι υπάρχει μία άλλη ταυτόχρονη διαγραφή, την οποία θα βοηθήσει να ολοκληρωθεί και έπειτα θα επανεκκινήσει ξανά από τη φάση αναζήτησης.

Ellen

Η τελευταία υλοποίηση παρουσιάζεται στο [12] και περιγράφει ένα non-blocking external δυαδικό δέντρο αναζήτησης. Είναι μία lock-free εκδοχή απλού δυαδικού δέντρου αναζήτησης (BST) που εφαρμόζει non-blocking συγχρονισμό χρησιμοποιώντας την ατομική εντολή compare-and-swap (CAS) στις λειτουργίες ενημέρωσης. Στους κόμβους του δέντρου υπάρχει ένα πεδίο που αναφέρται σαν "state", για να ανιχνεύονται ταυτόχρονες λειτουργίες ενημέρωσης. Το πεδίο αυτό σημειώνεται ανάλογα με τη λειτουργία

ενημέρωσης (εισαγωγή ή διαγραφή) που ενεργεί στον κόμβο όταν αυτήν ξεκινάει να εκτελείται. Μόλις αυτή τελειώσει το πεδίο "state" σημειώνεται ως "clean". Θέτοντας αυτό το πεδίο είναι σαν να κλειδώνουμε τους δείκτες προς τα παιδιά του κόμβου. Στις λειτουργίες που αλλάζουν τους δείκτες για τα παιδιά ενός κόμβου πρέπει να τιθεται το πεδίο "state" κατάλληλα. Παράδειγμα, η εισαγωγή εγγυάται ότι θα ολοκληρωθεί όταν αποκτήσει το κλείδωμα ενός κόμβου, δηλαδή θέσει κατάλληλα το πεδιο "state".

Σε αυτήν την υλοποίηση υπάρχει επίσης μία βοηθητική στρατηγική. Πιο συγκεκριμένα, μία λειτουργία βοηθάει μία άλλη να τελειώσει μόνο όταν η άλλη παρεμποδίζει τη δικιά της πρόοδο. Εφόσον η λειτουργία αναζήτησης δεν τροποποιεί τη δομή του δέντρου, δε μπλοκάρεται ποτέ και δε βοηθάει καμία άλλη λειτουργία. Ωστόσο, μία λειτουργία ενημέρωσης που προσπαθεί να κλειδώσει έναν κόμβο που είναι ήδη κλειδωμένος από μία άλλη λειτουργία βοηθάει την άλλη λειτουργία που κλείδωσε πρώτη τον κόμβο να ολοκληρωθεί και μετά συνεχίζει στις δικές τις αλλαγές στο δέντρο. Προκειμένου να επιτευχθεί αυτή η βοηθητική στρατηγική χρησιμοποιείται ένα νέο αντικείμενο, Info record. Όταν μία λειτουργία κλειδώνει έναν κόμβο αποθηκεύει αρκετές πληροφορίες στο αντικείμενο Info record, έτσι ώστε η άλλη λειτουργεία που ζητάει το ίδιο κλείδωμα να έχει αρκετές πληροφορίες για να τη βοηθήσει να ολοκληρώθεί. Ωστόσο, η χρήση αυτού του επιπλέον αντικειμένου Info record) προσθέτει επιπλέον κόστη διαχείρισης της δομής.

2.6.2 Λεπτομέρειες υλοποίησης

Αυτές οι πέντε υλοποιήσεις πολύπλοκων δέντρων αναζήτησης υλοποιήθηκαν σε C γλώσσα προγραμματισμού στο πλαίσιο της δημοσίευσης [8]. Ο κώδικας είναι διαθέσιμος στην ιστοσελίδα http://lpd.epfl.ch/site/ascylib.

2.7 Αξιολόγηση

2.7.1 Χαρακτηριστικά συστήματος αξιολόγησης

Το σύστημα στο οποίο αξιολογήσαμε τις υλοποιήσεις που παρουσιάστηκαν ήταν μία πλατφόρμα 60 πυρήνων, NUMA αρχιτεκτονική με τα παρακάτω χαρακτηριστικά.

- 4 sockets (Intel(R) Xeon(R) CPU E7-4880 v2 @ 2.50GHz)
- 15 πυρήνες ανά socket (30 νήματα με hyperthreading)
- 32KB L1 data cache ανά πυρήνα
- 32KB L1 instruction cache ανά πυρήνα
- 256KB L2 cache ανά πυρήνα
- 38MB L3 cache ανά socket
- 1TB RAM

2.7.2 Χαρακτηριστικά εκτέλεσης

Για την αξιολόγηση των υλοποιήσεων, εκτελούμε τυχαίες λειτουργίες για διαφορετικό αριθμό νημάτων, διαφορετικά εύρη κλειδιών και διαφορετικές αναλογίες αναζήτησης, εισαγωγής και διαγραφής. Πιο συγκεκριμένα:

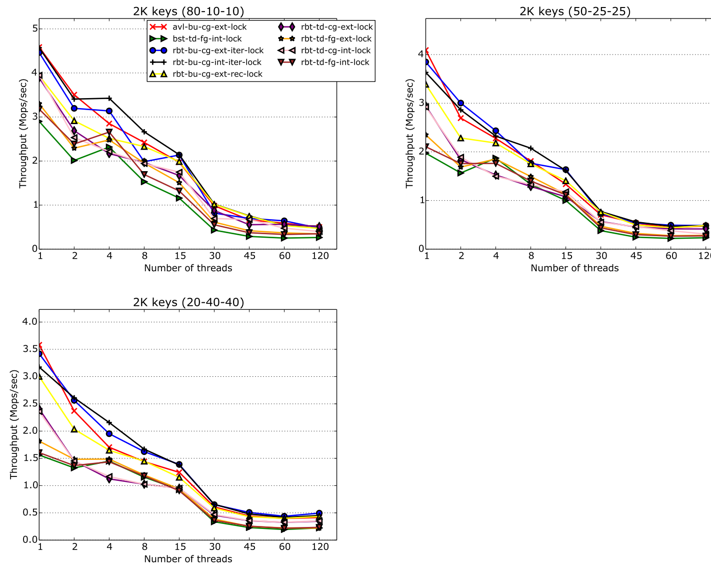
- Κάθε software νήμα "καρφιτσώνεται" σε hardware νήμα κατάλληλα, έτσι ώστε να εκμεταλλευτούμε την τοπικότητα ανάμεσα στα sockets. Για παράδειγμα, στην περίπτωση μικρού αριθμού νημάτων "καρφιτσώνουμε" όλα τα νήματα στο ίδιο socket για να διαμοιράζονται την ίδια L3 cache. Επίσης, πρώτα χρησιμοποιούμε όλους τους διαθέσιμους φυσικούς πυρήνες και έπειτα χρησιμοποιούμε hyperthreading.
- Η διάρκεια της εκτέλεσης είναι 5 δευτερόλεπτα, μέσα στην οποία κάθε νήμα εκτελεί τυχαίες λειτουργίες στη δομή, σύμφωνα με την αναλογία λειτουργιών που έχει επιλεχθεί.
- Το εύρος κλειδιών που επιλέγεται καθορίζει το μέγετος του δέντρου. Αξιολογήσαμε τις υλοποιήσεις των δέντρων αναζήτησης για εύρη κλειδιών 2K, 32K and 2000000. Στην αρχή κάθε εκτέλεσης το δέντρο αρχικοποιείται με το μισό πλήθος κόμβων με κλειδιά από το εύρος που έχει επιλεχθεί. Αυτό εγγυάται ότι κατά μέσο όρο οι μισές λειτουργίες θα είναι επιτυχείς και ότι ο χρόνος εκτέλεσης κάθε λείτουργίας θα παραμένει περίπου ίδιος κατά τη διάρκεια όλης της εκτέλεσης (το ποσοστό των εισαγωγών και το διαγραφών στη δομή είναι ίδιο, έτσι ώστε το δέντρο να έχει περίπου την ίδια μορφή σε όλη την εκτέλεση).
- Χρησιμοποιούμε τρεις διαφορετικές αναλογίες για τις λειτουργίες, 80-10-10 50-25-25 20-40-40, με 80%, 50% και 20% το ποσοστό λειτουργιών αναζήτησης στη δομή, ενώ το υπόλοιπο ποσοστό διαμοιράζεται ίσα σε λειτουργίες εισαγωγής και διαγραφής στη δομή.

2.7.3 Αποτελέσματα

Απλοϊκές υλοποιήσεις παράλληλων δέντρων αναζήτησης

Οι εκόνες 2.7, 2.8, 2.9 δίνουν την απόδοση των εκτελέσεων για τις απλοϊκές υλοποιήσεις παράλληλων δέντρων αναζήτησης στην πλατφόρμα που παρουσιάστηκε προηγουμένως. Τα δέντρα με εύρος κλειδιών 2K δεν κλιμακώνουν και η απόδοση τους μειώνεται όσο ο αριθμός των νημάτων αυξάνεται. Στα μικρά δέντρα εμφανίζονται πολλές συγκρούσεις, δηλαδή προσβάσεις στους ίδιους κόμβους από διαφορετικά νήματα και ως αποτέλεσμα υπάρχει υψηλή συμφόρηση. Καθώς το εύρος των κλειδιών επεκτείνεται (32K και 2000000), οι fine-grained υλοποιήσεις κλιμακώνουν μέχρι ένα μικρό αριθμό νημάτων μετά από τον οποίο η απόδοση καταρρέει. Πιο συγκεκριμένα, κλιμακώνουν μέχρι 8 νήματα. Πάνω από 15 νήματα, η απόδοση μειώνεται σημαντικά εξαιτίας της NUMA αρχιτεκτονικής, αφού τα νήματα πλέον "καρφιστώνονται" σε πάνω από ένα socket. Οπότε όταν εμφανιστεί ένα cache miss πρέπει να μεταφερθεί μία cache line από το ένα socket στο άλλο και αυτό έχει μεγάλο κόστος. Αντιθέτως, οι coarse-grained υλοποιήσεις δεν κλιμακώνουν

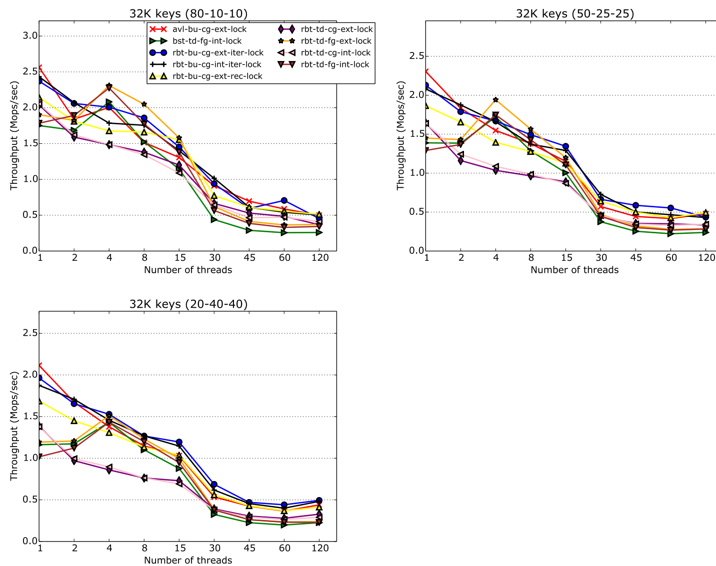
καθόλου, γιατί δεν παρέχουν παραλληλισμό. Η δομή προστατεύεται από ένα μοναδικό καθολικό κλείδωμα και ως αποτέλεσμα όλες οι προσβάσεις στο δέντρο σειριοποιούνται.



Σχήμα 2.7: Η απόδοση απλοϊκών παράλληλων δυαδικών δέντρων για εύρος κλειδιών 2K και για 3 διαφορετικές αναλογίες λειτουργιών.

Ανάμεσα στις fine-grained υλοποιήσεις η RBT external εκδοχή έχει τη μεγαλύτερη απόδοση. Αυτό οφείλεται στο ότι πρόκειται για ένα height-balance δέντρο σε σύκγριση με τη BST fine-grained υλοποίηση και εκτελεί γρηγορότερες διαγραφές συγκριτικά με τη RBT internal εκδοχή. Η διαγραφή στο internal fine-grained RBT δέντρο απαιτεί αποκλειστική πρόσβαση σε κάθε κόμβο μεταξύ αυτού που πρόκειται να διαγραφεί και του διάδοχού του (successor). Έτσι, ο κόμβος που πρόκειται να διαγραφεί παραμένει κλειδωμένος μέχρι να βρεθεί ο διάδοχος του. Με αυτόν τον τρόπο όλο το υπόδεντρο με ρίζα τον κόμβο που πρόκειται να διαγραφεί παραμένει κλειδωμένο και κανένα νήμα δε μπορεί να προχωρήσει σε αυτό. Αυτό έχει ως αποτέλεσμα οι διαγραφές να είναι πιο γρήγορες στα external tree τα οποία εμπλέκουν μόνο τους κόμβους-φύλλα. Από την άλλη, στις coarse-grained υλοποιήσεις η internal εκδοχή του δέντρου έχει καλύτερη απόδοση από την external. Στα internal δέντρα η λειτουργία αναζήτησης είναι γρηγορότερη, αφού μπορεί να τερματιστεί σε ένα μικρό βάθος δέντρου, ενώ στα external η λειτουργία τερματίζεται όταν φτάσουμε σε ένα φύλλο.

Στις top-down υλοποιήσεις καθώς διασχίζουμε ένα δέντρο από τη ρίζα στο κατάλληλο φύλλο, εκτελούνται και οι κατάλληλες τροποποιήσεις εκ των προτέρων για να εγγυάται ότι το δέντρο θα είναι balanced χωρίς να χρείαστει bottom-up διάσχιση. Αυτή η απαισιόδοξη φύση των top-down προσεγγίσεων έχει ως αποτέλεσμα να πραγματοποιούνται περισσότερες τροποποιήσεις και περιστροφές για κάθε λειτουργία συγκριτικά με την bottom-up υλοποιήση που εκτελεί μόνο τις απαραίτητες. Ως συνέπεια, οι top-down προσεγγίσεις

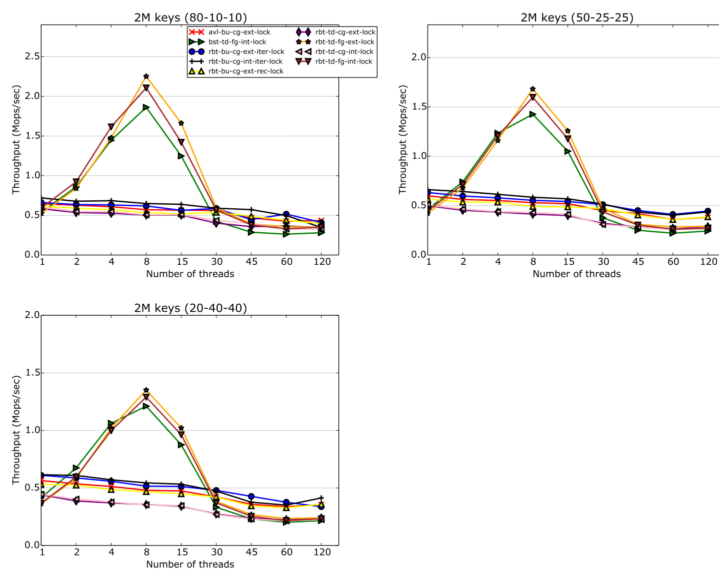


Σχήμα 2.8: Η απόδοση απλοϊκών παράλληλων δυαδικών δέντρων για εύρος κλειδιών 32K και για 3 διαφορετικές αναλογίες λειτουργιών.

έχουν χειρότερες αποδόσεις στις σειριακές εκτελέσεις (coarse-grained) και προκαλούν επιπλέον overhead. Στα απότελέσματά μας οι bottom-up coarse-grained υλοποιήσεις έχουν υψηλότερη απόδοση από τις top-down, καθώς το κόστος να διασχίσουμε δύο φορές το μονοπάτι κάνοντας τις απαιραίτητες μόνο τροποποιήσεις είναι μικρότερο από μία διάσχιση μονοπατιού που εκτελεί εκ των προτέρων τροποποιήσεις στο δέντρο για να μη χρειαστεί bottom-up διάσχιση. Τυπικά, περισσότερες αλλαγές-τροποποιήσεις εκτελούνται στις top-down προσεγγίσεις.

Συγκρίνοντας την bottom-up επαναληπτική (iterative) με την αναδρομική (recursive) υλοποίηση, η επαναληπτική υλοποίηση έχει καλύτερη απόδοση. Σε αυτό το δέντρο η bottom-up διάσχιση τερματίζεται όταν δεν απαιτείται πλέον rebalancing, ενώ στις αναδρομικές υλοποιήσεις η bottom-up φάση τερματίζεται στη ρίζα του δέντρου. Έτσι, τα αναδρομικά δέντρα διασχίζουν το μονοπάτι στον κατάλληλο κόμβος ακριβώς δύο φορές εκτελώντας τις κατάλληλες τροποποιήσεις στο δέντρο. Επιπλέον, υπάρχει επιπλέον overhead που σχετίζεται με τις αναδρομικές κλήσεις εξαιτίας της βαριάς χρήσης της στοίβας κατά την αναδρομή.

Τέλος, πρέπει να σημειώσουμε ότι στα μικρά δέντρα η απόδοση των coarse-grained υλοποιήσεων μειώνεται σημαντικά καθώς ο αριθμός των νημάτων αυξάνεται, σε αντίθεση με τα μεγαλύτερα δέντρα, όπου η απόδοση παραμένει περίπου ίδια για τους διαφορετικούς αριθμούς νημάτων. Αυτό συμβαίνει γιατί οι λειτουργίες στα μικρά δέντρα διαρκούν πολύ λίγο χρόνο και όσο μεγαλύτερος είναι ο αριθμός των νημάτων τόσο περισσότερα νήματα μπλοκάρουνται για να εκτελεστεί μία γρήγορη λειτουργία που στα μεγαλύτερα δέντρα διαρκεί περισσότερο. Ως αποτέλεσμα, οι λειτουργίες που εκτελούνται σε ενα σταθερό χρόνο στα μικρά δέντρα είναι αρκετά λιγότερες στην περίπτωση πολλαπλών νημάτων



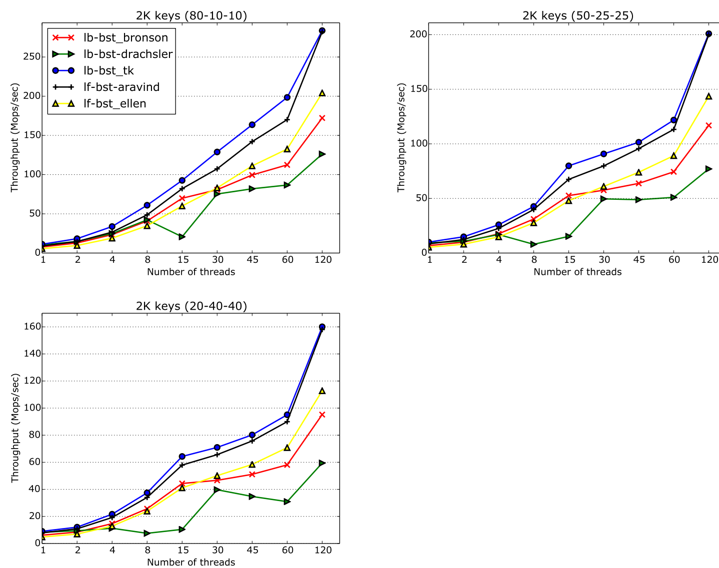
Σχήμα 2.9: Η απόδοση απλοϊκών παράλληλων δυαδικών δέντρων για εύρος κλειδιών 2000000 και για 3 διαφορετικές αναλογίες λειτουργιών.

συγκριτικά με ένα νήμα.

Πολύπλοκες υλοποιήσεις παράλληλων δέντρων αναζήτησης

Τα αποτελέσματα (Εικόνες 2.10, 2.11, 2.12) της αξιολογησής μας είναι αρκετά κοντά με αυτά που παρουσιάστηκαν στο [8]. Εξαιρώντας το BST Ticket δέντρο αναζήτησης, το Aravind παράλληλο δέντρο αναζήτησης έχει γενικά τη καλύτερη απόδοση. Αυτή η lock-free υλοποίηση χρησιμοποιεί δύο ατομικές εντολές κατά μέσο όρο για κάθε update λειτουργία και η οποία είναι κοντά σε ένα παράλληλο δέντρο αναζήτησης χωρίς συγχρονισμό σχετικά με τον αριθμό των store που γίνονται. Πιο συγκεκριμένα, αυτό το δέντρο παρουσιάζει χαμηλό παράθυρο συμφόρησης γιατί λειτουργεί σε επίπεδο ακμών (οι ακμές επισημαίνονται ως διαγραμμένες). Δεύτερον, αυτό το δέντρο δεσμεύει λιγότερα αντικείμενα και εκτελεί λιγότερες ατομικές εντολές σε αυτά ανα λειτουργία συγκριτικά με τους υπόλοιπους αλγορίθμους. Επίσης δε χρησιμοποιεί επιπλέον αντικείμενα για το συγχρονισμό των λειτουργίων (όπως π.χ η Info record δομή στον αλγόριθμο Ellen). Εκτελείται μία βοηθητική στρατηγική για τις διαγραφές και δεν υπάρχει βοηθητική στρατηγικη για τις εισαγωγές (insertions). Τέλος, πολλαπλά φύλλα μπορούν να διαγραφούν σε ένα βήμα του αλγορίθμου.

Συγκρίνοντας το BST Ticket παράλληλο δέντρο αναζήτησης με το Aravind δέντρο αναζήτησης, αυτά έχουν παρόμοια συμπεριφορά καθώς και τα δύο είναι πολύ κοντά σε ένα ασύγχρονο δέντρο αναζήτησης. Το BST Ticket δέντρο παρουσιάζει λίγο καλύτερη απόδοση από το Aravind δέντρο (Εικόνες 2.10 και 2.11 για μικρότερα δέντρα), αφού αυτό εκτελεί λιγότερες ατομικές εντολές ανά λειτουργία ενημέρωσης. Εκτελεί δύο ατομικές

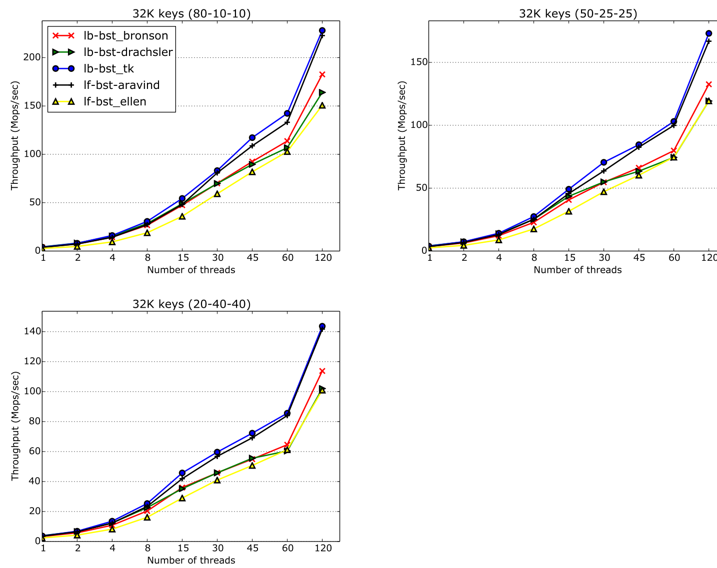


Σχήμα 2.10: Η απόδοση των πολύπολοκων παράλληλων δυαδικών δέντρων για εύρος κλειδιών 2K και για 3 διαφορετικές αναλογίες λειτουργιών.

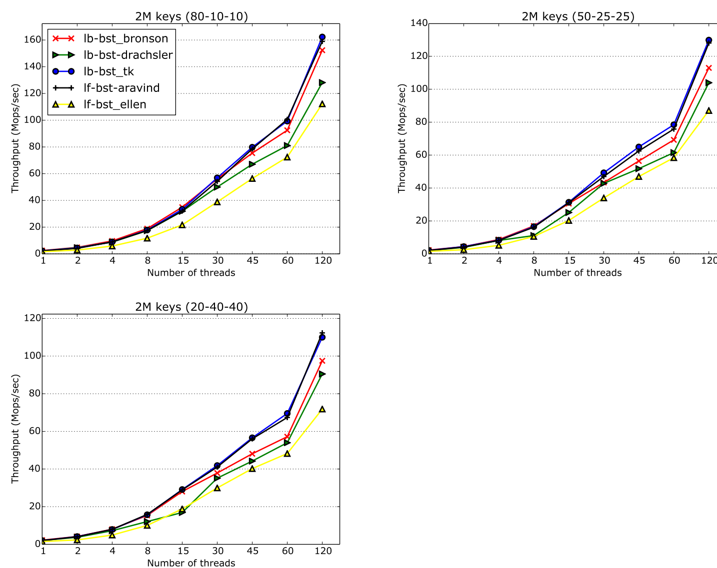
εντολές ανά διαγραφή, ενώ το Aravind δέντρο αναζήτησης εκτελεί τρεις ατομικές εντολές στη διαγραφή (και τα δύο εκτελούν μία ατομική εντολή ανά εισαγωγή).

Τα υπόλοιπα τρία δέντρα (Bronson, Drachsler, Ellen) έχουν τη χειρότερη απόδοση για όλα τα εύρη κλειδιών. Έχουν πιο πολύποκους μηχανισμούς συγχρονισμού. Το Ellen παράλληλο δυαδικό δέντρο αναζήτησης χρησιμοποιεί μία βοηθητική στρατηγική και για την εισαγωγή και για τη διαγραφή για τα στοιχεία που μία άλλη ταυτόχρονη λειτουργία θέλει να τροποποιήσει. Η βοηθητική στρατηγική είναι γενικά ακριβή και απαιτεί επιπλέον συγχρονισμό που προσθέτει επιπλέον overhead ανά λειτουργία. Επιπλέον, αυτή η υλοποίηση χρησιμοποιεί επιπλέον αντικείμενα (Info record) τα οποία αυξάνουν τον αριθμό των stores ανά λειτουργία και τον αριθμό των cache lines που μεταφέρονται. Το δεύτερο δέντρο, Draschler δέντρο αναζήτησης βασίζεται στα κλειδώματα και αποκτά ένα μεγάλο αριθμό κλειδωμάτων για κάθε λειτουγία ενημέρωσης κάτι που περιορίζει την απόδοση. Έχει καλύτερη απόδοση στις αναλογίες λειτουργιών όπου το ποσοστό αναζητήσεων είναι μεγαλύτερο και το κύριο πλεονέκτημα αυτής της υλοποίησης είναι ότι μπορεί να βρεθεί ο διάδοχος του κόμβου που πρόκειται να διαγραφεί με ένα μόνο βήμα ($\mathcal{O}(1)$) με τη βοήθεια του successor δείκτη της δομής του κόμβου. Τέλος, το Bronson παράλληλο δυαδικό δέντρο αναζήτησης εμφανίζει καλύτερη απόδοση από τα δύο προηγουμενα (Ellen, Draschler) στα περισσότερα πειράματά μας (Εικόνες 2.11 και 2.12) επειδή είναι ισοζυγισμένο. Είναι ένα relaxed AVL δέντρο. Ωστόσο, χρησιμοποιεί ομοίως ένα πολύπλοκο μηχανισμό κλειδωμάτων και την τεχνική hand-over-hand και τα νήματα μπλοκάρονται για αρκετή ώρα περιμένοντας μία λειτουργία ενημέρωσης να ολοκληρωθεί. Ως αποτέλεσμα, η απόδοσή του είναι πολύ χαμηλή για μικρά δέντρα όπου υπάρχει υψηλή συμφόρηση (Εικόνα 2.10).

Όπως εξηγείται στο [8] η συνάφεια μνήμης είναι ο πιο περιοριστικός παράγοντας



Σχήμα 2.11: Η απόδοση των πολύπολοκων παράλληλων δυαδικών δέντρων για εύρος κλειδιών 32K και για 3 διαφορετικές αναλογίες λειτουργιών.



Σχήμα 2.12: Η απόδοση των πολύπολοκων παράλληλων δυαδικών δέντρων για εύρος κλειδιών 2000000 και για 3 διαφορετικές αναλογίες λειτουργιών.

στους παράλληλους αλγορίθμους αναζήτησης στα πολυπύρηνα συστήματα, καθώς ο αριθμός των μεταφορών των cache lines αυξάνεται καθώς ο αριθμός των νημάτων αυξάνεται. Ως εκ τούτου, οι περισσότεροι παράλληλοι αλγόριθμοι δέντρων αναζήτησης προσπαθούν

να ελαχιστοποιήσουν τις μεταφορές cache lines που απαιτούνται σε μία λειτουργία κάτι το οποίο είναι άμεσα συνδεδεμένο με τον αριθμό των stores στη δομή. Τα stores προκαλούν invalidation των cache lines και σύμφωνα με το πρωτόκολλο συνάφειας το αποτέλεσμα είναι αύξηση των cache misses. Ο γενικός κανόνας είναι ότι όσο λιγότερα cache misses προκαλεί ο αλγόριθμος, τόσο καλύτερα κλιμακώνει.

Τέλος, τα αποτελέσματα μας αποδεικνύουν ότι οι lock-based και οι lock-free αλγόριθμοι είναι αρκετά κοντά στην απόδοση, αλλά στην περίπτωση υψηλού αριθμού νημάτων (υψηλή συμφόρηση) οι lock-free υλοποιήσεις παρέχουν καλύτερη κλιμακωσιμότητα. Μάλιστα όσο ο αριθμός των stores μίας επιτυχούς λειτουργίας πρέπει να είναι πιο κοντά σε έναν ασύγχρονο, σειριακό αλγόριθμο. Όσο πιο κοντά στο σειριακό αλγόριθμο είναι μία υλοποίηση τόσο υψηλότερη απόδοση έχει.

Κεφάλαιο 3

Transactional Memory

3.1 Transactional Memory (TM)

Μεταβαίνοντας από τα μονοεπεξεργαστικά συστήματα στα πολυεπεξεργαστικά δημιουργήθηκε η ανάγκη για εύρεση non-blocking μηχανισμών που θα έδιναν καλύτερη κλιμακωσιμότητα και θα απλοποιούσαν τον παράλληλο προγραμματισμό. Αυτό οδήγησε στο Transactional Memory (TM). Το πιο σημαντικό πλεονέκτημά του είναι ότι δεν υπάρχουν κλειδώματα και δεν οδηγούμαστε σε αδιέξοδα.

Με το transactional memory ένα σύνολο εντολών load και store εκτελούνται με ατομικό τρόπο. Σύνθετες εντολές μπορούν να εκτελούνται ταυτόχρονα σε απομόνωση από τις άλλες και είτε να ολοκληρωθούν είτε όχι, όπως μία δοσοληψία. Οι προγραμματιστές είναι εξοικειωμένοι με αυτήν την τεχνολογία από τις βάσεις δεδομένων. Η δοσοληψία είναι μία μονάδα εργασίας εκτελείται ατομικά.

Η ιδέα πίσω από το transactional memory είναι ότι δεν υπάρχει ανάγκη συγχρονισμού αλλά το σύστημα μπορεί να ανιχνεύσει ότι μία σύγκρουση έχει εμφανιστεί εξαιτίας της παράλληλης εκτέλεσης εργασιών σε πολλαπλούς πυρήνες. Μία σύγκρουση συμβαίνει όταν δύο δοσοληψίες εκτελούν μία λειτουργία στην ίδια θέση μνήμης. Υπάρχουν δύο τύποι συκγρούσεων:

1. Μία δοσοληψία γράφει σε μία θέση μνήμης στην οποία μία άλλη διαβάζει ή γράφει, επίσης.
2. Μία δοσοληψία διαβάζει μία θέση μνήμης στην οποία μία άλλη εκτελεί μια εγγραφή.

Αν δεν ανιχνευθούν συγκρούσεις κατά τη διάρκεια της δοσοληψίας, το TM σύστημα προσπαθεί να καταστήσει μόνιμα τα αποτελέσματα της δοσοληψίας και να ενημερώσει όλους τους επεξεργαστές για τις αλλαγές της δοσοληψίας. Όλες οι αλλαγές γίνονται ορατές και μόνιμες. Αυτό είναι ένα **transactional commit**. Διαφορετικά, αν ανιχνευθούν συγκρούσεις κατά τη διάρκεια εκτέλεσης της δοσοληψίας το TM σύστημα αναιρεί την τρέχουσα δοσοληψία, απορρίπτει όλες τις αλλαγές που έκανε και επαναφέρει το σύστημα στην αρχική του κατάσταση σαν η δοσοληψία να μην είχε αρχίσει ποτέ. Αυτό είναι ένα

transactional abort. Η ταχύτητα με την οποία ανιχνεύονται οι συγκρούσεις και εκτελούνται τα commits/aborts είναι ο πιο περιοριστικός παράγοντας στην απόδοση του TM συστήματος.

Το TM σύστημα χωρίζεται σε τρεις κατηγορίες, το Software Transactional Memory (STM), το Hardware Transactional Memory (HTM) και το Hybrid Transactional Memory.

3.1.1 Software Transactional Memory (STM)

Το software transactional memory υλοποιείται αποκλειστικά σε επίπεδο software. Δεν υπάρχει η χρήση hardware υλικού στην ανίχνευση συγκρούσεων ή στην εκτέλεση commit/abort. Μπορεί να υλοποιηθεί σαν lock-free αλγόριθμος ή να χρησιμοποιήσει κλειδώματα και είναι μία software βιβλιοθήκη. Κάποιες STM υλοποιήσεις είναι: TiniSTM (C), STMNet (C#), CL-STM (Common Lisp), STM Library (Haskell), Deuce, DSTM2 (Java), ScalaSTM (Scala). Τα κύρια πλεονέκτήματά του είναι ότι μπορεί να χρησιμοποιηθεί σε οποιαδήποτε πλατφόρμα (σύστημα), αφού δε χρειάζεται υποστήριξη από το υλικό, επιτρέπει την υλοποίηση πολύπλοκων αλγορίθμων και μπορεί να τροποποιηθεί εύκολα. Ωστόσο, επειδή είναι υλοποιημένο αποκλειστικά σε software είναι σχετικά αργό και με μικρή απόδοση. Η ανίχνευση συγκρούσεων και η επαναφορά του συστήματος στην αρχική του κατάσταση στην περίπτωστη transactional abort είναι δαπανηρές και χρονοβόρες.

3.1.2 Hardware Transactional Memory (HTM)

Το hardware transactional memory προτάθηκε ως βελτίωση του STM. Η υλοποίηση του HTM γίνεται αποκλειστικά στο υλικό (hardware). Η ανίχνευση των συγκρούσεων και εκτέλεση των transactional commit/abort εκτελούνται στο υλικό. Στόχος του είναι η μείωση του χρόνου με τον οποίο επιβαρύνεται το σύστημα κατά την εκτέλεση του TM.

Αν και το HTM δεν είναι τόσο χρονοβόρο όσο το STM, συνεχίζει να προσθέτει κάποιο κόστος στην εκτέλεση της δοσοληψίας ειδικά στην περίπτωση διαδοχικών aborts. Το μεγαλύτερο όμως μειονέκτημα είναι οι περιορισμένοι πόροι του υλικού που μπορεί να οδηγήσουν σε αδυναμία εκτέλεσης της δοσοληψίας. Για παράδειγμα, το μέγεθος της δοσοληψίας περιορίζεται από το μέγεθος της μνήμης. Έπίσης, ένας άλλος περιορισμός είναι η χρήση επεκτάσεων στο σύνολο των εντολών του HTM. Ο κώδικας του προγράμματος πρέπει να ξαναγράφεται για κάθε διαφορετικό επεξεργαστή που υποστηρίζει διαφορετική HTM υλοποίηση χρησιμοποιώντας διαφορετικές επεκτάσεις κάθε φορά. Τέλος, είναι προφανές ότι ένα πρόγραμμα που χρησιμοποιεί μία HTM υλοποίηση δε μπορεί να εκτελεστεί σε μία πλατφόρμα που δεν υποστηρίζει HTM.

3.1.3 Hybrid Transactional Memory

Αυτή η υλοποίηση είναι ένας συνδυασμός των δύο προηγούμενων μοντέλων και εκμεταλλεύεται τα πλεονεκτήματα και από τα δύο. Για παράδειγμα, στα [13], [14] έχουν προταθεί hybrid TM που είναι υλοποιήσεις στο software, έτσι ώστε να χρησιμοποιήσουν το HTM για να αυξήσουν την απόδοσή τους, αλλά αυτές οι υλοποιήσεις εξαρτώνται από

το HTM. Στα [15] και [16] έχουν προταθεί προσέγγισεις οι οποίες χρησιμοποιούν το υλικό για να επιταχύνουν TM υλοποίησεις που ελέγχονται από software (hardware accelerated STMs).

3.2 Βασικά χαρακτηριστικά TM συστημάτων

Διαφορετικές υλοποιήσεις TM συνδυάζουν διαφορετικά χαρακτηριστικά. Τα βασικά χαρακτηριστικά των TM υλοποιήσεων είναι:

- **Data versioning:** Ο μηχανισμός με τον οποίο το TM σύστημα διαχειρίζεται τα writes των ταυτόχρονων δοσοληψιών.
  - *Eager versioning (undo-log based):* Η μνήμη ενημερώνεται άμεσα και σε περίπτωση abort οι παλιές τιμές κρατιούνται σε ένα "undo log".
  - *Lazy versioning (write-buffer based):* Οι τιμές (writes) μιας δοσοληψίας γράφονται σε έναν buffer και όταν η δοσοληψία κάνει commit ενημερώνεται η μνήμη με αυτές.
- **Conflict detection:** Ο τρόπος με τον οποίο ανιχνεύονται τα conflicts.
  - *Pessimistic detection:* Κάθε φορά που γίνεται ένα load ή store, γίνεται έλεγχος για conflict.
  - *Optimistic detection:* Τα conflicts ελέγχονται όταν γίνεται το commit.
- **Conflict resolution:** Η πολιτική επίλυσης των conflict, όταν αυτά ανιχνεύονται.
- **Isolation:** Όταν τα conflicts μπορούν να ανιχνευθούν και σε non-transactional κώδικα έχουμε *strong isolation*. Όταν conflicts ανιχνεύονται μόνο σε transactional mode έχουμε *weak isolation*.
- **Granularity:** Η μονάδα στην οποία το TM σύστημα ανιχνεύει τα conflicts.
- **Best effort:** Αυτό το χαρακτηριστικό εμφανίζεται μόνο σε πραγματικά HTM συστήματα. Χρησιμοποιώντας transactional mode το σύστημα δε μπορεί να εγγυηθεί πρόοδο. Μία δοσοληψία μπορεί να αποτυγχάνει συνεχώς και για αυτό απαιτείται ένα non-transactional εναλλακτικό μονοπάτι (fallback path).
- **Conflicts:** Μία δοσοληψία μπορεί να αποτύχει για διάφορους λόγους.
  - *Data conflict*
  - *Capacity abort*
  - *Explicit abort*
  - *Other*

3.3 Intel's Haswell HTM

Σε αυτήν την ενότητα θα αναλύσουμε μία HTM υλοποίηση που προσφέρει η Intel μέσα από τους επεξεργαστές Haswell. Η Intel το 2013 ανακοίνωσε ότι η Haswell αρχιτεκτονική της θα προσφέρει υποστήριξη υλικού για transactional memory και ο Haswell έγινε ο πρώτος επεξεργαστής με HTM.

Για την αξιοποίηση του TM ο προγραμματιστής πρέπει να επισημαίνει την περιοχή του κώδικα που πρέπει να εκτελεστεί ως transaction. Όσο εκτελείται αυτό το κομμάτι του κώδικα σαν δοσοληψία, το σύστημα βρίσκεται σε transactional mode. Σε transactional mode όλες οι θέσεις μνήμης που ζητάει η δοσοληψία μεταφέρονται στη μνήμη cache και κατηγοριοποιούνται σε δύο σετ, το read και το write set. Το read set αποτελείται από όλες τις θέσεις μνήμης που διαβάζει το transaction και το write από όλες εκείνες στις οποίες γράφει το transaction.

Καθώς εκτελείται η δοσοληψία μπορεί να ανιχνευθεί μία σύγκρουση. Τότε η δοσοληψία αποτυγχάνει (abort) και το HTM αναιρεί τις αλλαγές που έκανε και επαναφέρει το σύστημα στην αρχική του κατάσταση. Αυτό υλοποιείται αρκετά εύκολα, καθώς όλες οι αλλαγές της δοσοληψίας αποθηκέυονται στην L1D μνήμη (ή/και στην L2) και η κύρια μνήμη δεν έχει ενημερωθεί για τις αλλαγές. Ως αποτέλεσμα, το TM ακυρώνει όλες τις transactional cache lines της L1D μνήμης και η κύρια μνήμη παραμένει αμετάβλητη. Αν δεν ανιχνευθεί καμία σύγκρουση η δοσοληψία κάνει commit και όλες οι τροποποιημένες transactional cache lines μεταφέρονται στην κύρια μνήμη.

3.3.1 Transactional Synchronizations Extensions (TSX)

Για την αξιοποίηση του HTM το σύνολο της αρχιτεκτονικής (x86) έχει επεκταθεί με νέες εντολές που δίνουν τη δυνατότητα πειραματισμού με το HTM. Για την ευκολότερη χρήση του HTM προσφέρονται συναρτήσεις σε γλώσσα προγραμματισμού C/C++ που ονομάζονται Transactional Synchronization Extensions (TSX). Ο προγραμματιστής έχει στη διάθεσή του δύο διεπαφές (software interfaces). Η πρώτη καλείται Hardware Lock Elision (HLE) και η δεύτερη Restricted Transactional Memory (RTM), που είναι πιο ολοκληρωμένη υλοποίηση. Η RTM απαιτεί από τον προγραμματιστή να παρέχει έναν εναλλακτικό κώδικα (fallback path) σε περίπτωση που η εκτέλεση του κρίσιμου τμήματος σε transactional mode δεν είναι επιτυχημένη.

Η βασική διαφορά μεταξύ του HLE και του RTM είναι ότι το λογισμικό που είναι γραμμένο χρησιμοποιώντας το HLE μπορεί να τρέξει σε ένα σύστημα που δεν υποστηρίζει TSX. Σε αυτή την περίπτωση το κρίσιμο τμήμα θα εκτελεστεί χρησιμοποιώντας απευθείας ένα κλέιδωμα. Αντιθέτως, το λογισμικό που είναι γραμμένο σε RTM δε μπορεί να εκτελεστεί σε επεξεργαστή που δεν υποστηρίζει TSX. Ωστόσο, το RTM προσφέρει περισσότερη ευελιξία σχετικά με τις ενέργειες που μπορούν να πραγματοποιηθούν μετά από ένα transactional abort. Ο προγραμματιστής μπορεί να ορίσει την εντολή και το κομμάτι κώδικα που θα εκτελεστεί στην περίπτωση του abort (fallback path). Για να χρησιμοποιήσουμε TSX πρέπει ο μεταγλωττιστής να υποστηρίζει gcc-4.8x (ή πιο σύγχρονη έκδοση) και να εισάγει στον κώδικά του τη βιβλιοθήκη "immintrin.h" ή αν διαθέτει πιο παλία έκδοση gcc, πρέπει να εισαχθεί στο πρόγραμμα η βιβλιοθήκη "rtm.h".

Hardware Lock Elision (HLE)

Το hardware lock elision είναι ένας εύκολος τρόπος να χρησιμοποιούμε την transactional memory σε ήδη υπάρχων κώδικα. Η βασική ιδέα του HLE είναι να αφαιρέσει τα κλει-

δώματα και να εκτελέσει το κρίσιμο τμήμα ως δοσοληψία χρησιμοποιώντας το HTM. Αν ανιχνευθεί κάποια σύγκρουση η δοσοληψία αποτυγχάνει (abort) και το κρίσιμο τμήμα επανεκτελείται με χρήση κλειδώματος. Αν δεν ανιχνευθεί κάποια σύγκρουση η δοσοληψία επιτυγχάνει (commit) και οι αλλαγές γίνονται ορατές και μόνιμες. Το HLE είναι απλό αλλά μας περιορίζει επειδή χρησιμοποιούνται κλειδώματα. Το listing 3.1 παρουσιάζει ένα παράδειγμα χρήσης του HLE σε C γλώσσα προγραμματισμού.

Listing 3.1: Παράδειγμα χρήσης HLE

```
/* Traditional lock implementation */
/* acquire lock */
while(__sync_lock_test_and_set(&lock_var) == 0)
 /* do nothing */;
... Critical section with lock acquired ...
/* release lock */
__sync_lock_release(&lock_var);


/* HLE implementation */
/* elide lock */
while(__hle_acquire_test_and_set(&lock_var) == 0)
/* do nothing */;
... Critical section with lock acquired ...
/* release lock */
__hle_release_clear(&lock_var);
```

Restricted Transactional Memory (RTM)

Το restricted transactional memory (RTM) είναι μία εναλλακτική υλοποίηση από το HLE που δίνει περισσότερη ευελιξία στον προγραμματιστή να καθορίσει το fallback path που θα εκτελεστεί αν η δοσοληψία αποτύχει. Υπάρχουν τέσσερις νέες εντολές, XBEGIN, XEND, XTEST και XABORT. Ο προγραμματιστής επισημαίνει το μέρος του κώδικα που θέλει να εκτελεστεί ατομικά (κρίσιμο τμήμα) χρησιμοποιώντας τις εντολές XBEGIN και XEND. Οι εντολές XBEGIN και XEND επισημαίνουν την αρχή και το τέλος του κρίσιμου τμήματος αντίστοιχα. Όταν μία διεργασία/νήμα φτάσει στην εντολή XEND στον κώδικα, η δοσολοψία κάνει commit και η μνήμη ενημερώνεται για τις αλλαγές που έκανε. Η εντολή XTEST επιστρέφει 1 αν η διεργασία/νήμα βρίσκεται σε transactional mode και διαφορετικά 0 και με την εντολή XABORT(status) ο προγραμματιστής μπορεί ρητά να κάνει abort μια δοσοληψία. Η μεταβλητή status χρησιμοποιείται για να καθορίσει το λόγο του transactional abort. Μία τέτοια ρητή διακοπή της δοσοληψίας είναι χρήσιμη όταν ο προγραμματιστής θέλει να καθορίσει την αποτυχία μιας δοσοληψίας.

Αν μια σύγκρουση ανιχνευθεί κατά τη διάκεια της δοσοληψίας, πιθανώς θα υπάρξει ματαίωσή της. Μετά από ένα transactional abort ο fallback handler είναι υπέυθυνος για το ποια εντολή θα εκτελέσει η διεργασία/νήμα για να συνεχίσει την εκτέλεση. Ο

προγραμματιστής ορίζει τη θέση μνήμης στον κώδικα που θα εκτελεστεί σε περίπτωση transactional abort (fallback address). Αυτή η διεύθυνση είναι ακριβώς η επόμενη εντολή μετά την XBEGIN. Η XBEGIN επιστρέφει μία τιμή που δείχνει αν μία διεργασία/νήμα είναι σε transactional mode ή έχει γίνει aborted. Ως αποτέλεσμα, ο EAX καταχωρητής ενημερώνεται σύμφωνα με την κατάσταση της δοσοληψίας (σχήμα 3.1). Ο προγραμματιστής μπορεί να ελέγξει αν η δοσοληψία έχει ξεκινήσει ή αν έχει γίνει abort εκτελώντας μία λογική πράξη μεταξύ της τιμής που επιστρέφεται και των παρακάτω σταθερών:

- _XBEGIN_STARTED: Η δοσοληψία έχει ξεκινήσει επιτυχώς.
- _XABORT_CONFLICT: Η δοσοληψία έχει γίνει abort λόγω σύγκρουσης με τη μνήμη ενός άλλου νήματος/διεργασίας.
- _XABORT_CAPACITY: Η δοσοληψία έχει γίνει abort λόγω υπερχείλισης στη μνήμη.
- _XABORT_EXPLICIT: Η δοσοληψία διακόπηκε μετά από ρητή εντολή από τον προγραμματιστή.
- _XABORT_RETRY: Η δοσοληψία διακόπηκε αλλά αν ξαναπροσπαθήσει μπορεί να επιτύχει.
- _XABORT_DEBUG: Η δοσοληψία διακόπηκε λόγω debug.
- _XABORT_NESTED: Transactional abort σε μία εσωτερική εμφωλευμένη δοσοληψία.

Θέση bit στον EAX καταχωρητή	**Σημασία**
0	Είναι στο λογικό 1 αν το abort έγινε από την εντολή XABORT.
1	Αν είναι στο λογικό 1, τότε η δοσοληψία είναι πιθανό να επιτύχει σε επόμενη προσπάθεια.
2	Είναι στο λογικό 1 αν ένας λογικός επεξεργαστής ήρθε σε σύγκρουση με μία διεύθυνση μνήμης της δοσοληψίας.
3	Είναι στο λογικό 1 αν κάποιος εσωτερικός buffer υπερχείλισε.
4	Είναι στο λογικό 1 αν εντοπίστηκε κάποια debug διακοπή.
5	Είναι στο λογικό 1 αν συνέβη abort σε μία εμφωλιασμένη δοσοληψία.
23:6	Reserved.
31:24	Το argument που δόθηκε στην εντολή XABORT (είναι έγκυρο όταν το 0 είναι στο λογικό 1).

Σχήμα 3.1: Η κατάσταση της δοσοληψίας αποτυπώνεται στα bit του EAX καταχωρητή.

Κάποιες αιτίες abort μπορεί να οδηγούν πάντα σε αποτυχία της δοσοληψίας. Ως αποτέλεσμα, κάθε φορά που εκτελούμε το κρίσιμο τμήμα σε transactional mode χρησιμοποιώντας το HTM, η δοσοληψία θα αποτογχάνει συνέχεια και δε θα υπάρχει πρόοδος στο πρόγραμμά μας. Για παράδειγμα, αν το read ή το write set μιας δοσοληψίας ξεπερνούν το μέγεθος της μνήμης, η δοσοληψία θα αποτυγχάνει συνεχώς λόγω capacity abort. Σε αυτην την περίπτωση ο προγραμματιστής πρέπει να ορίσει έναν back-off μηχανισμό όπως το fallback path που έχει αναφερθεί. Το fallback path είναι μία εναλλάκτική υλοποίηση κώδικα που δε χρησιμοποιεί το RTM και είναι συνήθως ένα κομμάτι κώδικα με coarse-grained locking. Αυτή η υλοποίηση είναι απαραίτητη για την εγγύηση προόδου στην εκτέλεση του προγράμματος.

Το fallback path υλοποιείται σαν ένα coarse-grained locking τμήμα κώδικα που έχει ένα καθολικό κλείδωμα. Στην περίπτωση αυτή πρέπει το καθολικό κλείδωμα να εισάγεται στο read set των δοσοληψιών, για να γίνονται abort όταν αυτό κλειδωθεί από μία άλλη διεργασία/νήμα και να μην εμφανιστούν προβλήματα συνάφειας της μνημής. Το σχήμα 3.2 παρουσιάζει ένα παράδειγμα εκτέλεσης κατά το οποίο δύο νήματα προσπαθούν να εκτελέσουν τον ίδιο κώδικα ταυτόχρονα. Το πρώτο νήμα μπαίνει στο κρίσιμο τμήμα αποκτώντας το καθολικό κλείδωμα και το δεύτερο χρησιμοποιώντας την RTM υλοποίηση. Σε αυτό το παράδειγμα το δεύτερο νήμα δε θα ανιχνεύσει σύγκρουση και η δοσοληψία θα κάνει commit και θα ενημερώσει τη μεταβλητή count, ενώ το πρώτο νήμα δε θα ενημερωθεί για αυτήν την αλλαγή. Αυτό είναι ένα πρόβλημα συνάφειας της μνήμης. Κατά συνέπεια, ο προγραμματιστής είναι υπέυθυνος να προσθέσει το καθολικό κλείδωμα στο read set της δοσοληψίας. Αυτό επιτυγχάνεται με ένα απλό read της τιμής του κλειδώματος. Αν στην αρχή της δοσοληψίας το καθολικό κλείδωμα είναι κλειδωμένο από ένα άλλο νήμα, ο προγραμματιστής πρέπει να κάνει ρητά abort τη δοσοληψία (explicit abort). Το listing 3.2 αποτελεί ενα παράδειγμα κατά το οποίο ένα pthread_spinlock προστίθεται στο read set της δοσοληψίας μέσω ενός απλού read της τιμής του και σε περίπτωση που αυτό δεν είναι ελεύθερο η δοσοληψία αποτυγχάνει ρητά μέσω της εντολής XABORT.

Listing 3.2: Παράδειγμα: προσθήκη του καθολικού κλειδώματος στο read set της δοσοληψίας

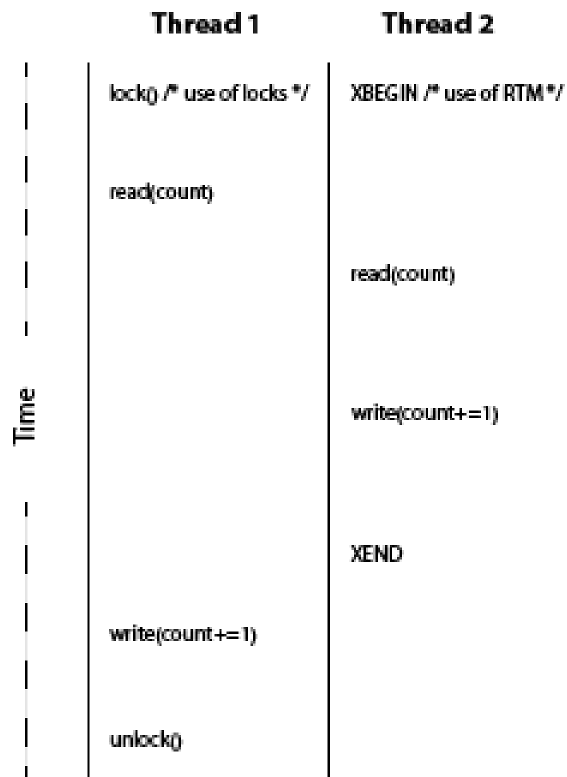
```
if ((int)spin_lock != 1)
                    _xabort();
if (pthread_mutex_t.__data.__lock != 0)
                            _xabort ();
```

Τέλος, το listing 3.3 περιγράφει ένα παράδειγμα χρήσης του RTM σε C γλώσσα προγραμματισμού.

Listing 3.3: Ένα RTM παράδειγμα

```
int aborts = MAX_TX_RETRIES;
lock_t = fallback_global_lock;
start_tx:
        int status = TX_BEGIN();
        if(status == TX_BEGIN_STARTED){
                if (fallback_global_lock is locked)
                        TX_ABORT();
        ... Critical Section ...
        TX_END();
        }else{ /* status != TX_BEGIN_STARTED */
                if (--aborts > 0)
                        /* retry transaction */
                        goto start_tx;
                acquire_lock(fallback_global_lock);
                ... Critical Section ...
```

```
        release_lock(fallback_global_lock);
}
```



Σχήμα 3.2: Μία παράλληλη εκτέλεση δύο νημάτων με RTM που οδηγεί σε προβλήματα συνάφειας της μνήμης.

Κεφάλαιο 4

Παραλληλοποιώντας τον αλγόριθμο του Dijkstra

4.1 Ο αλγόριθμος του Dijkstra

4.1.1 Ο αλγορίθμος

Ο αλγόριθμος του Dijkstra είναι ένας greedy αλγόριθμος που υπολογίζει το ελάχιστο μονοπάτι από μία μοναδική πηγή προς όλους τους κόμβους όταν οι ακμές είναι μη αρνητικές. Ο αλγόριθμος βασίζεται στην παρατήρηση ότι κάθε μικρότερο μονοπάτι ενός μεγαλύτερου ελάχιστου μονοπατιού είναι και το ίδιο βέλτιστο (ελάχιστο) μονοπάτι. Το μήκος του μονοπατιού είναι το άθροισμα των βαρών όλων των ακμών που συμμετέχουν σε αυτό.

Ο αλγόριθμος εξετάζει τις κορυφές σε αύξουσα σειρά από την απόστασή τους από την πηγή. Κάθε φορά που διερευνάται ένας κόμβος αυτός έχει πάρει την ελάχιστη (βέλτιστη) απόσταση από την πηγή. Ο αλγόριθμος κατασκευάζει το δέντρο ελαχίστων μονοπατιών. Σε κάθε βήμα προστίθεται μία νέα ακμή που αντιστοιχεί στην κατασκευή του ελάχιστου μονοπατιού για έναν καινούργιο κόμβο. Η νέα ακμή προστίθεται στο ελάχιστο μονοπάτι για το νέο κόμβο, αν το νέο μονοπάτι από την πηγή προς τον κόμβο είναι μικρότερο από την προηγούμενη απόσταση από την πηγή προς τον κόμβο. Η διαδικασία κατά την οποία υπολογίζεται μία εκτίμηση της απόστασης από την πηγή προς τον κόμβο και ενημερώνεται αυτή η απόσταση με τη νέα μικρότερη τιμή ονομάζεται **relaxation**. Οι πίνακες dist και pred αποθηκεύουν την βέλτιστη (ελάχιστη) τιμή από την πηγή για κάθε κόμβο και τον αμέσως προηγούμενο κόμβο στο βέλτιστο μονοπάτι, αντίστοιχα.

Για ένα γράφο $G = (V, E)$, όπου V είναι ένα σετ από κορυφές και E είναι ένα σετ από ακμές, ο αλγόριθμος έχει δύο σετ από κορυφές: το S σετ που είναι κορυφές οι οποίες έχουν πάρει την ελάχιστη απόσταση τους από την πηγή και το σετ $V \backslash S$ που είναι οι κορυφές που δεν έχουν εξερευνηθεί ακόμα. Ο αλγόριθμος έχει τα εξής βήματα:

1. Αρχικά το σετ S είναι άδειο.

2. Αρχικοποίηση όλων των αποστάσεων των κορυφών από την πηγή με άπειρο (∞) και την απόσταση της ίδιας της πηγής με 0.
3. Ενώ υπάρχουν ακόμα κορυφές στο σετ $V\backslash S$ (ανεξερεύνητες)

   (α) Διάλεξε την ανεξερεύνητη κορυφή u από το σετ $V\backslash S$ η οποία έχει την ελάχιστη απόσταση από την πηγή.
   (β) Τοποθέτησε την κορυφή u στο σετ S.
   (γ) Για κάθε ακμή της κορυφής, κάνε τα κατάλληλα relaxations για τους κόμβους που είναι ακόμα στο σετ $V\backslash S$. Για κάθε γειτονική κορυφή v, αν η απόσταση της κορυφής u προστιθέμενη με το βάρος της ακμής u-v είναι μικρότερη από την απόσταση που έχει το v, ενημέρωσε την απόσταση αυτήν.

Το listing 4.1 είναι ένας ψευδοκώδικας του αλγορίθμου του Dijkstra. Το Q σύνολο είναι το σετ των ανεξερεύνητων ακμών ($V\backslash S$) το οποίο είναι υλοποιημένο σαν ουρά προτεραιότητας και οι πίνακες dist, prev αποθηκεύουν την αποστάση από την πηγή για ένα κόμβο v και τον προηγούμενο κόμβο από το v στο βέλτιστο μονοπάτι από την πηγή, αντίστοιχα. Η συνάρτηση add_with_priority() προσθέτει ένα στοιχείο στην ουρά με μία συσχετιζόμενη προτεραιότητα (την ελάχιστη απόσταση από την πηγή), η συνάρτηση extract_min() αφαιρεί ένα στοιχείο από την ουρά προτεραιότητας, το στοιχείο με τη μεγαλύτερη προτεραιότητα (ελάχιστη απόσταση από τη πηγή στο τρέχον βήμα) και επιστρέφει αυτό το στοιχείο και η συνάρτηση decrease_priority() ενημερώνει την προτεραιότητα ενός στοιχείου στην ουρά.

Listing 4.1: Ο αλγόριθμος του Dijsktra

```
function dijkstra(graph, source):
        dist[source] ← 0 //Initialization

        create vertex set Q //Set of unvisited vertices

        for each vertex v in Graph:
                if v ≠ source
                        dist[v] ← INFINITY //Unknown distance from source to v
                        prev[v] ← UNDEFINED //Predecessor of v

                Q.add_with_priority(v, dist[v])


        while Q is not empty: //The main loop
                u ← Q.extract_min() //Remove and return best vertex
                for each neighbor v of u: //Only if v that is still in Q
                        sum = dist[u] + length(u, v)
                        if sum < dist[v] //A shorter path has been found
                                dist[v] ← sum
                                prev[v] ← u
                                Q.decrease_priority(v, sum)

        return dist[], prev[]
```

4.1.2 Η πολυπλοκότητα του αλγορίθμου

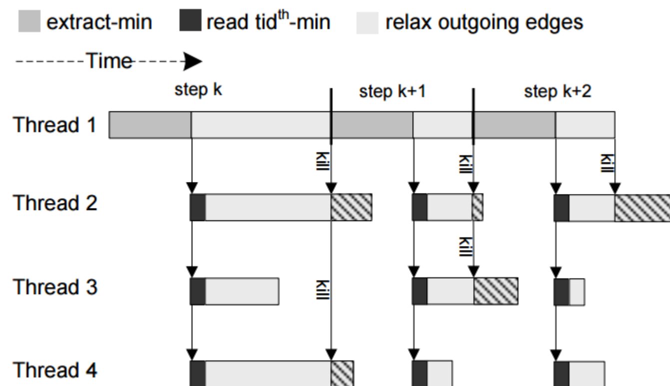
Η απλούστερη υλοποίηση του αλγορίθμου αποθηκεύει το σετ των κορυφών σαν μία συνηθισμένη λίστα ή πίνακας. Η extract_min() συνάρτηση παίρνει χρόνο $\mathcal{O}(V)$ και γίνονται |V| τέτοιες λειτουργίες. Έτσι, ο συνολικός χρόνος για την extract_min() σε όλο το while loop είναι $\mathcal{O}(V^2)$. Αφού ο συνολικός αριθμός των ακμών σε όλη τη λίστα γειτνίασης είναι |E|, το for loop θα επαναληφθεί |E| φορές και κάθε επανάληψη παίρνει χρόνο $\mathcal{O}(1)$. Έτσι, η πολυπλοκότητα του αλγορίθμου με μία συνηθισμένη συνδεδεμένη λίστα ή υλοποίηση με πίνακα είναι $\mathcal{O}(V^2 + E) = \mathcal{O}(V^2)$.

Για αραιούς γράφους (γράφοι με μικρό αριθμό ακμών συγκριτικά με τον αριθμό των κορυφών), ο αλγόριθμος μπορεί να έχει καλύτερη πολυπλοκότητα αποθηκεύοντας το σετ κορυφών σαν μία λίστα γειτνίασης και χρησιμοποιώντας ένα ήμι-ισοζυγισμένο δυαδικό δέντρο αναζήτησης, ένα δυαδικό σωρό ή ένα Fibonacci heap σαν ουρά προτεραιότητας. Αυτό έχει ως αποτέλεσμα μία πιο αποτελεσματική υλοποίηση της extract_min() συνάρτησης. Η extract_min() παίρνει χρόνο $\mathcal{O}(\log V)$ και γίνονται |V| τέτοιες λειτουργίες. Η συνάρτηση decrease_priority() (ή decrease_key()) παίρνει χρόνο $\mathcal{O}(\log V)$ στην περίπτωση ενός ημι-ισοζυγισμένου δυαδικού δέντρου αναζήτησης ή ενός δυαδικού σωρού και $\mathcal{O}(1)$ στην περίπτωση ενός fibonacci heap για κάθε μία από τις |E| το πλήθος ακμές. Κατά συνέπεια, ο χρόνος εκτέλεσης του αλγορίθμου με ένα ημι-ισοζυγισμένο δυαδικό δέντρο αναζήτησης ή ένα δυαδικό σωρό είναι ανάλογος με $\mathcal{O}((E+V)\log V)$ και με ένα fibonacci heap γίνεται $\mathcal{O}(E + V\log V)$.

4.2 Μία τεχνική παραλληλοποίησης στον αλγόριθμο του Dijkstra

Πριν περιγράψουμε μία προτεινόμενη παραλληλοποίηση πάνω στον αλγόριθμο του Dijkstra, θα εισάγουμε την έννοια των Helper Threads. Τα helper threads είναι μία τεχνική βελτιστοποίησης που χρησιμοποιείται για παραλληλισμό με σκοπό να επιταχύνει ένα πρόγραμμα και να παρέχει υψηλή απόδοση. Ουσιαστικά, είναι "βοηθητικά" νήματα που εκτελούν συγκεκριμένους υπολογισμούς εκ μέρους ενός κυρίως νήματος (main thread) με στόχο να βοηθήσουν το main thread και να μειώσουν τις εργασίες του. Τυπικά, εκμεταλλευόμαστε αυτήν την τεχνική για να προφορτώσουμε μελλοντικά δεδομένα στα οποία θα γίνει πρόσβαση ή να υπολογίσουμε το αποτέλεσμα κομματιών του κώδικα που διαφορετικά θα εκτελούνταν από το main thread.

Αυτή η ενότητα περιγράφει μία παραλληλοποίηση του αλγορίθμου του Dijkstra που παρουσιάζεται στις δημοσιεύσεις [20], [21]. Τα δύο πιο σημαντικά προβλήματα του αλγορίθμου είναι ότι ένα πολύ μικρό κομμάτι του μπορεί να παραλληλοποιηθεί και ότι πρέπει να υπάρξει πολύ καλός συγχρονισμός. Για να αντιμετωπίστουν αυτά τα δύο προβλήματα η προτεινόμενη παραλληλοποίηση του αλγορίθμου χρησιμοποιεί την έννοια των Helper Threads για να εξάγει περισσότερο παραλληλισμό και το μηχανισμό του Transactional Memory ως μέσο συγχρονισμού των ταυτόχρονων προσβάσεων στις μοιραζόμενες δομές δεδομένων.



Σχήμα 4.1: Σχήμα εκτέλεσης του αλγορίθμου.

Η ιδέα είναι ότι τα helper threads θα ξεφορτώνουν λειτουργίες από το main thread. Πιο συγκεκριμένα, το main thread εκτελεί πολλά relaxations των κόμβων-στοιχείων της ουράς προτεραιότητας. Έτσι, παράλληλα βοηθητικά νήματα μπορούν ταυτόχρονα να εκτελούν relaxations από κόμβους. Καθώς το main thread εξάγει και ενημερώνει τους γείτονες του στοιχείου που είναι πρώτο στην ουρά προτεραιότητας, k helper threads μπορούν να ενημερώνουν τους γείτονες από τους k επόμενους κόμβους στην ουρά προτεραιότητας με σκοπό να ξεφορτώσουν πράξεις από το main thread στην επόμενή του επανάληψη.

Οι κόμβοι που αναλαμβάνουν τα helper threads βρίσκονται στις πρώτες k θέσεις στην ουρά και ίσως έχουν ήδη πάρει τη βέλτιστη τιμή τους (ελάχιστη απόσταση) από την πηγή με μεγάλη πιθανότητα. Έτσι, όταν τα helper threads εκτελέσουν τα relaxations για τις ακμές τους, οι γείτονες που αντιστοιχούν σε αυτές τις ακμές ίσως αποκτήσουν και αυτοί τις βέλτιστες τιμές τους. Ως αποτέλεσμα, στην επόμενη επανάληψη το main thread θα ελέγξει αυτούς τους γείτονες και δε θα εκτελέσει κανένα relaxation. Από την άλλη πλευρά, τα helper threads ίσως εκτελέσουν realaxations για έναν κόμβο που δεν έχει αποκτήσει τη βέλτιστη τιμή του ακόμα. Σε αυτή την περίπτωση ο κόμβος θα αποκτήσει την βέλτιστη τιμή του όταν τελικά εξεταστεί από το main thread αργότερα. Οπότε η ορθότητα του αλγορίθμου δεν επηρεάζεται.

Το main thread εκτελείται όπως στη σειριακή εκδοχή του αλγορίθμου. Σε κάθε επανάληψη εξάγει τον ελάχιστο κόμβο (κορυφή) από την ουρά προτεραιότητας και εκτελεί τα relaxations που αντιστοιχούν σε αυτόν. Ταυτόχρονα, το k-οστό helper thread διαβάζει την απόσταση του k-οστού κόμβου της ουράς προτεραιότητας και προσπαθεί να κάνει τα κατάλληλα relaxations για τις εξερχόμενες ακμές του σύμφωνα με αυτήν την τιμή της απόστασης. Όταν το main thread τελείωσει όλα τα relaxations του, ειδοποιεί τα helper threads να σταματήσουν και όλα τα νήματα μαζί προχωρούν στην επόμενη επανάληψη. Αυτό το σχήμα απεικονείζεται στην εικόνα 4.1 *.

Στην περίπτωση που τα helper threads αναγκάζονται από το main thread να σταμα-

* Η εικόνα λαμβάνετα από το [21].

τήσουν τους υπολογισμούς τους και να προχωρήσουν στην επόμενη επανάληψη, είναι πιθανό κάποια από αυτά να μην έχουν ενημερώσει κάποιους από τους γείτονες του κόμβου, αφήνοντας τους με τις παλιές αποστάσεις τους. Αυτό δεν αποτελεί πρόβλημα καθώς όλοι οι γείτονες του κόμβου θα αποκτήσουν τελικά τις βέλτιστες τιμές τους όταν ο κόμβος φτάσει στην κορυφή της ουράς προτεραιότητας και εξεταστεί από το main thread.

Ο κώδικας που εκτελεί το main και τα helper threads φαίνεται στα listings 4.2 και 4.3 σε C γλώσσα προγραμματισμού, αντίστοιχα. Σε κάθε επανάληψη το main thread εξάγει τον κόμβο με την υψηλότερη προτεραιότητα (ελάχιστη απόσταση από την πηγή) από την ουρά προτεραιότητας. Ταυτόχρονα, τα helper threads περιμένουν (κολλημένα στο while loop) μέχρι το main να τελειώσει την εξαγωγή του κόμβου. Ακολούθως, κάθε helper thread διαβάζει χωρίς να εξάγει (ReadMin λειτουργία γραμμές 6-7 στον κώδικα των helper threads) ένα από τους πρώτους k κόμβους στην ουρά. Όπως ήδη εξηγήθηκε, τα helper threads ξεφορτώνουν υπολογισμούς από το main thread και έτσι η γραμμή 28 στο listing 4.2 θα υπολογιστεί ως true λιγότερες φορές και το main thread δε θα χρειαστεί να εκτελέσει τις λειτουργίες των γραμμών 29-31.

Το προτεινόμενο σχήμα πρέπει να παρέχει ατομικότητα επειδή μπορεί να εμφανιστεί conflict όταν δύο ή περισσότερα νήματα ενημερώνουν ταυτόχρονα τον ίδιο γείτονα, ή διαφορετικούς γείτονες αλλά αλλάζουν το ίδιο μέρος της ουράς. Για να πετύχουμε ατομικότητα πρέπει οι ενημερώσεις στην ουρά μέσω της DecreaseKey() συνάρτησης, καθώς επίσης και οι ενημερώσεις στους πίνακες dist και pred να εσωκλείονται σε μία δοσοληψία τόσο για το main όσο και για τα helper threads. Με αυτόν τον τρόπο, όταν εμφανιστεί ένα conflict, μόνο ένα νήμα θα μπορέσει να προχωρήσει, να κάνει commit τη δοσοληψία και να ενημερώσει την ουρά, ενώ τα υπόλοιπα θα πρέπει να επαναλάβουν τους υπολογισμούς τους.

Όπως ήδη αναφέρθηκε, όταν το main thread τελειώσει τα relaxations που πρέπει να κάνει, ειδοποιεί τα helper threads να σταματήσουν και προχωρούν όλα τα νήματα μαζί στην επόμενη επανάληψη. Για να υλοποιηθεί αυτό, ο αλγόριθμος χρησιμοποιεί transactional memory (TM). Πιο συγκεκριμένα, όταν το main thread ολοκληρώσει την επανάληψη του εσωτερικού loop για τα relaxations, θέτει τη μεταβλητή "done" στο 1. Αυτό σημαίνει ότι το main thread θα προχωρήσει στην επόμενη επανάληψη του εξωτερικού while loop για τον επόμενο κόμβο και αναγκάζει επίσης τα helper threads να σταματήσουν και να ακολουθήσουν, τερματίζοντας τους υπολογισμούς. Αφού τα helper threads είναι σε transactional mode και η μεταβλητή "done" είναι στο read set τους, αυτά θα γίνουν aborted και θα ξαναπροσπαθήσουν τη δοσοληψία. Ωστόσο, όταν τα helper threads προσπαθήσουν να εκτελέσουν νέα δοσοληψία, θα βρουν με μεγάλη πιθανότητα τη μεταβλητή "done" στην τιμή 1 και θα σταματήσουν τα relaxations τους για τους υπόλοιπους γείτονες και θα προχωρήσουν σε νέα επανάληψη του εξωτερικού while loop. Αν το main thread εκτελέσει την ExtractMin() πολύ γρήγορα και θέσει τη μεταβλητή "done" πίσω στο 0, τα helper threads θα χάσουν αυτήν την τελευταία ειδοποίηση και θα συνεχίσουν από το σημείο που σταμάτησαν. Αυτό δεν επηρεάζει την ορθότητα του αλγορίθμου. Τα helper threads ίσως ενημερώσουν τις αποστάσεις των γειτόνων με υποβέλτιστες τιμές. Αυτές όμως θα επαναγραφτούν με τις νέες βέλτιστες τιμές όταν οι κορυφές που εξετάστηκαν από τα helper threads φτάσουν στην κορυφή της ουράς προτεραιότητας.

Ο σκοπός του αλγορίθμου είναι να χρησιμοποιηθούν τα helper threads μόνο για να ξεφορτώσουν υπολογισμούς από το main thread και όχι να παρεμποδίσουν την πρόοδό του. Επιπλέον, αυτό το σχήμα προσπαθεί να ελαχιστοποιήσει το χρόνο που ξοδεύεται στο συγρονισμό και στα transactional aborts. Τα helper threads εκτελούν λειτουργίες στην ουρά, παρεμβαίνοντας όσο το δυνατόν λιγότερο στις εργασίες του main thread, ακόμα και αν αυτά δεν κάνουν χρήσιμη δουλειά. Χρησιμοποιώντας το TM σύστημα πρέπει να υπάρξει μία πολιτική επίλυσης συγκρούσεων που να ευνοεί το main thread και να ελαχιστοποιεί το overhead των transactional aborts του main thread.

Ο αλγόριθμος αξιολογήθηκε σε ένα πραγματικό HTM σύστημα (Intel's Haswell HTM). Η ουρά προτεραιότητας υλοποιείται με δυαδικό σωρό (binary heap), χρησιμοποιώντας υλοποίηση με πίνακα. Η ReadMin εκτελείται από τα helper threads σε σταθερό χρόνο ($\mathcal{O}(1)$) και οι ExtractMin() και DecreaseKey() λειτουργίες παίρνουν χρόνο $\mathcal{O}(\log n)$. Όπως φαίνεται στους κώδικες για τα main και τα helper threads υλοποιήθηκε μία coarse-grained δοσοληψία για το main thread. Στον αρχικό αλγόριθμο που προτείνεται στις δημοσιεύσεις [20] και [21] υπάρχει μία δοσοληψία για κάθε ακμή (πιθανό relaxation) που εξετάζεται. Αυτό έχει ως αποτέλεσμα πολλές μικρές δοσοληψίες ειδικά στους πυκνούς γράφους και ως συνέπεια εμφανίζεται επιπρόσθετο overhead που σχετίζεται με την έναρξη και τον τερματισμό πολλών διαδοχικών δοσοληψιών. Έτσι, εμείς εξετάζουμε περισσότερες από μία ακμές μέσα σε μία δοσοληψία. Αναμένουμε αυτήν η πιο coarse-grained προσέγγιση να οδηγήσει σε καλύτερο speedup σε μικρούς γράφους και σε πιθανά capacity aborts στους μεγάλους γράφους. Για αυτό, πρέπει να βρούμε μία λύση η οποία μετριάζει το overhead από την ύπαρξη πολλών μικρών δοσοληψιών και το κόστος από πολλά διαδοχικά transactional aborts.

Όπως εξηγήθηκε στο προηγούμενο κεφάλαιο ο πιο περιοριστικός παράγοντας για την κλιμακωσιμότητα είναι το φαινόμενο false sharing. Διαφορετικά νήματα ίσως τροποποιούν διαφορετικά μέρη της δομής που μοιράζονται την ίδια cache line. Επειδή στο πραγματικό HTM σύστημα που χρησιμοποιήσαμε οι συγκρούσεις (conflicts) ανιχνεύονται σε επίπεδο cache line, η περίπτωση που διαφορετικά νήματα ενημερώνουν σε transactional mode ανεξάρτητα στοιχεία που μοιράζονται την ίδια cache line θα οδηγήσει σε transactional aborts. Για να αποφύγουμε τέτοιες καταστάσεις χρησιμοποιούμε την τεχνική structure padding σε όλες τις μοιραζόμενες δομές όπως η ουρά προτεραιότητας, οι πίνακες dist και pred, έτσι ώστε τα διαφορετικά στοιχεία αυτών των δομών να βρίσκονται σε διαφορετικές cache lines.

Τέλος, το πραγματικό HTM σύστημα που χρησιμοποιήσαμε για την αξιολόγηση είναι μία best effort υλοποίηση. Συνεπώς, ένα fallback path είναι απαραίτητο. Χρησιμοποιούμε ένα καθολικό κλείδωμα, μοιραζόμενο ανάμεσα σε όλα τα νήματα για να προστατεύσουμε το κρίσιμο τμήμα. Ωστόσο, όταν ένα νήμα αποκτά το καθολικό κλείδωμα (global lock), τα υπόλοιπα θα υφίστανται transactional abort, αφού έχουν το καθολικό κλείδωμα στο read σετ τους και θα συνεχίσουν να αποτυγχάνουν μέχρι να ελευθερωθεί το κλείδωμα. Ο σκόπος του αλγορίθμου είναι να εκμεταλλευτεί την έννοια των helper thread, έτσι ώστε το main thread να εκτελέσει λιγότερα relaxations και όχι να καθυστερείται η πρόοδος του. Το main thread πρέπει να τρέχει σχεδόν στην ταχύτητα της σειριακής εκτέλεσης. Για να υλοποιήσουμε αυτήν την πολιτική που ευνοεί το main thread, το καθολικό κλείδωμα

μπορεί να αποκτηθεί μόνο από το main thread. Τα helper threads δεν παίρνουν ποτέ το καθολικό κλείδωμα και εκτελούν όλους τους υπολογισμούς τους στα κοινά δεδομένα με δοσοληψίες. Έτσι, μπορεί να αποτυγχάνουν συνεχώς και δεν υπάρχει εγγύηση προόδου για τα helper threads.

Listing 4.2: Ο κώδικας του main thread για ένα πραγματικό HTM σύστημα.

```
while(heap->curr_size > 0){

        my_min = bh_extract_min(heap);
        done = 0;

        /* Find the id of the vertex. */
        my_min_id = my_min->vertex_id;

        /* Read the key (weight) of my vertex.*/
        my_min_key = dist[my_min_id].value;

        if(my_min_key < INFINITY){

                /* adjacency list for neighbors */
                v = g->adj[my_min_id];

                if(v != NULL){
                        while(1){

                                /* Check neighbors for relaxation. */
                                begin_transaction(num_retries, &fallback_lock, tid);
                                for(i=0; i< num_neighbr; i++){

                                        distv = dist[v->id].value;
                                        sum = my_min_key + v->weight;

                                        /* Relax */
                                        if(distv > sum){
                                                decrease_key_mt(heap, v->id, sum);
                                                pred[v->id].value = my_min_id;
                                                dist[v->id].value = sum;
                                        }
                                        v = v->next;
                                        if(v == NULL)
                                                break;
                                }
                                end_transaction(&fallback_lock, counter);
                                if(v == NULL)
                                        break;
                        }
                }
                done=1;
        }
}
```

Listing 4.3: Ο κώδικας των helper threads για ένα πραγματικό HTM σύστημα.

```
while(heap->curr_size > 0){

        while(done == 1);

        /* ReadMin */
        my_min_id = heap->node_array[tid].vertex_id;
```

```
        my_min_key = dist[my_min_id].value;

        if(my_min_key < INFINITY){

                /* Check neighbors for relaxation. */
                for(v=g->adj[my_min_id]; v!=NULL && !done; v=v->next){


                        if(begin_transaction(num_retries, &fallback_lock, tid) != -1){
                                if(done == 0){
                                        distv = dist[v->id].value;
                                        sum = my_min_key + v->weight;

                                        if(distv > sum){
                                                decrease_key_mt(heap, v->id, sum);
                                                pred[v->id].value = my_min_id;
                                                dist[v->id].value = sum;
                                        }
                                }else
                                        _xabort(0xaa);
                        }else
                                break;
                        end_transaction(&fallback_lock, counter);

                }
        }
}
```

4.3 Χαρακτηριστικά συστήματος

Το σύστημα που χρησιμοποιήσαμε για την αξιολογήση της προτεινόμενης τεχνικής παραλληλοποίησης του αλγορίθμου του Dijkstra είναι μία 28-πύρηνη πλατφόρμα, NUMA αρχιτεκτονική με τα παρακάτω χαρακτηριστικά.

- 2 sockets (Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz)
- 14 πυρήνες ανά socket (28 νήματα με hyperthreading)
- 32KB L1 data cache ανά πυρήνα
- 32KB L1 instruction cache ανά πυρήνα
- 256KB L2 cache ανά πυρήνα
- 35MB L3 cache ανά socket
- 128GB RAM
- Hardware Transactional Memory:
  - lazy data versioning
  - eager conflict resolution
  - best effort HTM
  - strong isolation
  - cache line granularity
  - 4MB read set

– 22KB write set

Στο κομμάτι της αξιολόγησης, κάθε software thread καρφιτσώνεται χειροκίνητα σε ένα hardware thread, έτσι ώστε να εκμεταλλευτούμε την τοπικότητα των sockets. Πρώτα καρφιτσώνουμε software threads, έτσι ώστε να γεμίσει το πρώτο socket (τα πρώτα 14 νήματα) και να μοιράζονται την ίδια κοινή L3 cache. Και μετά καρφιτσώνουμε νήματα στο δεύτερο socket του μηχανήματός μας. Η αξιολόγησή μας αποκαλύπτει ότι το NUMA effect επηρεάζει αρνητικά την κλιμακωσιμότητα. Σε περίπτωση cache miss η μεταφορά μιας διεύθυνσης μνήμης από το ένα socket στο άλλο έχει μεγάλο κόστος.

4.4 Αξιολόγηση

4.4.1 Αξιολόγηση του σειριακού αλγορίθμου

Αρχικά, αξιολογήσαμε το σειριακό αλγόριθμο. Πιο συγκεκριμένα, εκτελέσαμε στην παραπάνω πλατφόρμα τον κώδικα του main thread (απλός αλγόριθμος Dijkstra) για γράφους διαφορετικών μεγεθών, τόσο πυκνούς όσο και αραιούς γράφους. Σε αυτές τις εκτελέσεις δε χρειάζεται να εκτελούνται relaxations μέσα σε transaction, γιατί πρόκειται για σειριακή εκτέλεση. Ο στόχος αυτών των εκτελέσεων είναι να εκτιμήσουμε το παράλληλισμό που μπορεί να εξαχθεί από τον κατ'εξοχην σειριακό αλγόριθμο του Dijkstra.

Χωρίσαμε τον αλγόριθμο σε 4 φάσεις και μετρήσαμε το χρόνο εκτέλεσης της καθεμίας. Η πρώτη φάση είναι η ExtractMin() συνάρτηση που παίρνει χρόνο ανάλογο με $\mathcal{O}(\log n)$, η δεύτερη είναι η φάση υπολογισμών για την εκτίμηση της απόστασης (γραμμή 9 στο listing 4.4) από την πηγή, η τρίτη είναι η DecreaseKey() λειτουργία και η τέταρτη οι ενημερώσεις της απόστασης και του προηγούμενου κομβου στο μονοπάτι στους πίνακες dist και pred (γραμμές 18-19 στο listing 4.4). Το σχήμα 4.2 παρουσιάζει τα αποτελέσματά μας για δύο πυκνούς γράφους (έναν rmat γράφο με 1M κορυφές και 100M ακμές, και έναν random γράφο με 10M κορυφές και 500M ακμές) και έναν αραιό γράφο (rmat με 100M κορυφές και 100M ακμές).

Listing 4.4: Οι 4 φάσεις του αλγορίθμου.

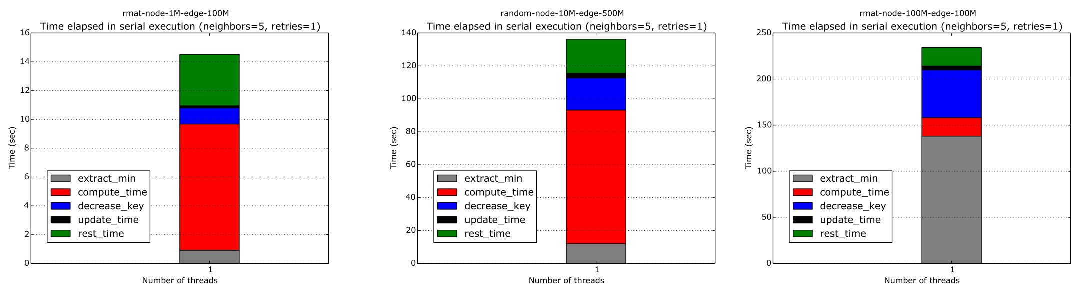
```
while Q not empty do
        start_timer(extract_min)
        u ← ExtractMin(Q);
        stop_timer(extract_min)

        done ← 0;
        foreach v adjacent to u do
                start_timer(compute_time)
                sum ← d[u] + w(u, v);
                stop_timer(compute_time)

                Begin−Transaction
```

```
                    if d[v] > sum then
                            start_timer(decrease_key)
                            DecreaseKey(Q, v, sum);
                            stop_timer(decrease_key)
                            start_timer(update_time)
                            d[v] ← sum;
                            p[v] ← u;
                            stop_timer(update_time)
                    End−Transaction
            end

            Begin−Transaction
            done ← 1;
            End−Transaction
    end
```



Σχήμα 4.2: Αξοιλογήση των 4 φάσεων του σειριακού αλγορίθμου σε 3 διαφορετικούς γράφους.

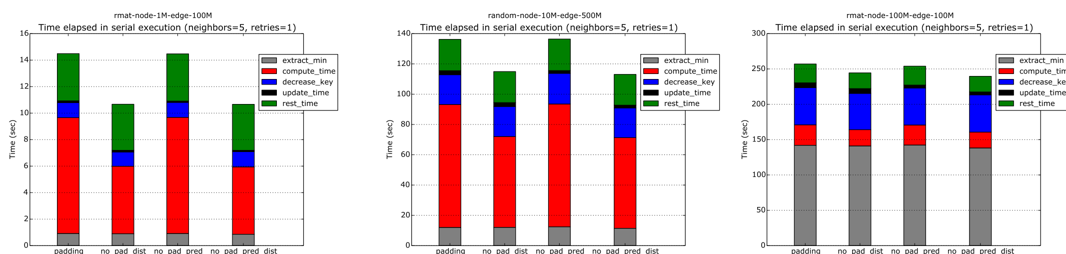
Στο προτεινόμενο σχήμα παραλληλοποίησης μόνο η decrease_key και η update φάση μπορούν να παραλληλισθούν. Τα helper threads εκτελούν κάποιους υπολογισμούς, έτσι ώστε το if branch του main thread (γραμμή 13 στο listing 4.4) να εκτελεστεί ως true λιγότερες φορές. Σύμφωνα με το σχήμα 4.2 ο κλάδος αυτός είναι περίπου το 12-25% του συνολικού χρόνου εκτέλεσης του main thread. Έτσι, το main thread μπορεί να κερδίσει ένα πολύ μικρό ποσοστό του συνολικού χρόνου εκτέλεσης και αυτή η παραλληλοποίηση μπορεί να προσφέρει μικρή απόδοση (speedup). Οι extract_min και compute φάσεις του αλγορίθμου πρέπει να εκτελούνται και από το main thread για όλους τους κόμβους και για όλες τις ακμές. Ως αποτέλεσμα, συμπεραίνουμε ότι ο αλγόριθμος του Dijkstra είναι ένας δύσκολα παραλληλοποιήσιμος αλγόριθμος και το μεγαλύτερο μέρος του αλγορίθμου είναι σειριακό.

Στον rmat-node-100M-edge-100M γράφο η extract_min φάση αποτελεί ένα μεγάλο μερίδιο του συνολικού χρόνου εκτέλεσης, συγκριτικά με τους άλλους 2 γράφους. Η ExtractMin() συνάρτηση διαρκεί χρόνο ανάλογο του $\mathcal{O}(\log n)$. Κατα συνέπεια, όσο περισσότερους κόμβους έχει ένας γράφος, τόσο μεγαλύτερη είναι η δομή binary heap και

τόσο περισσότερο διαρκεί η ExtractMin() συνάρτηση. Αυτό το συμπέρασμα απεικονίζεται στο γράφο rmat-node-100M-edge-100M, που είναι ένας πολύ μεγάλος γράφος. Στο ίδιο συμπέρασμα καταλήγουμε και για την DecreaseKey() συνάρτηση, καθώς και αυτή διαρκεί χρόνο ανάλογο του $\mathcal{O}(\log n)$.

Από την άλλη πλευρά, στους άλλους δύο γράφους (rmat-node-10M-edge-500M και random-node-1M-edge-100M) η compute φάση είναι η πιο χρονοβόρα φάση του αλγορίθμου. Υπολογίζει τη νέα απόσταση που συνδέεται με μία ακμή και επαναλαμβάνει αυτήν την διαδικασία για όλες τις ακμές του γράφου. Ως αποτέλεσμα, όσο πιο πυκνός είναι ο γράφος, τόσο περισσότερο διαρκεί αυτή η φάση. Ο γράφος rmat-node-100M-edge-100M είναι ένας αραιός γράφος με αναλογία ακμών προς κόμβους 1, οπότε κάθε επανάληψη του εξωτερικού while loop απαιτεί μόνο μία επανάληψη κατά μέσο όρο της compute φάσης, ενώ στους πιο πυκνούς γράφους αυτή επαναλαμβάνεται περισσότερες φορές σε κάθε επανάληψη του while loop. Ως εκ τούτου, η compute φάση είναι πιο χρονοβόρα στους πυκνούς γράφους και δεν αποτελεί μεγάλο μέρος του συνολικού χρόνου εκτέλεσης στους αραιούς γράφους.

Στη σειριακή εκτέλεση δεν υπάρχουν κατά κύριο λόγο conflict aborts, αφού υπάρχει μόνο ένα νήμα. Χρησιμοποιήσαμε την τεχνική του padding με σκοπό να αποφύγουμε το φαινόμενο false sharing που θα οδηγούσε σε data conflicts στην περίπτωση πολλαπλών νημάτων. Επομένως, οι δομές του αλγορίθμου όπως ο d[] και ο p[] πίνακας δε χρειάζεται να είναι padded στη σειριακή εκτέλεση. Αξιολογήσαμε το σειριακό αλγόριθμο για διαφορετικούς γράφους χωρίς να χρησιμοποιούμε την τεχνική padding στις δομές και τα αποτελέσματά μας φαίνονται στο σχήμα 4.3.



Σχήμα 4.3: Αξιολόγηση της τεχνικής structure padding στο σειριακό αλγόριθμο.

Τα απότελέσματά μας δείχνουν ότι ακόμα και όταν δε χρησιμοποιούμε padding στις δομές, η αναλογία των 4 φάσεων του αλγορίθμου στο συνολικό χρόνο εκτέλεσης παραμένει η ίδια. Το μοτίβο των διαφορετικών φάσεων είναι ίδιο σε όλες τις εκτελέσεις. Ωστόσο, παρατηρούμε ότι όταν δε χρησιμοποιούμε padding ο συνολικός χρόνος εκτέλεσης μειώνεται σημαντικά, ειδικά όταν αφαιρούμε το padding από τον πίνακα d[], ο οποίος χρησιμοποιείται περισσότερο στην compute φάση που είναι η πιο χρονοβόρα για τους πυκνούς γράφους. Χωρίς το padding, διαδοχικά στοιχεία αποθηκεύονται στην ίδια cache line. Έτσι, το main thread μπορεί να βρει κάποια στοιχεία στην cache μνήμη του και να αποφύγει μεταφορές cache lines από την κύρια μνήμη σε κάθε read/write λειτουργία. Επιπλέον, χωρίς το padding τα capacity misses μειώνονται, καθώς μία cache line

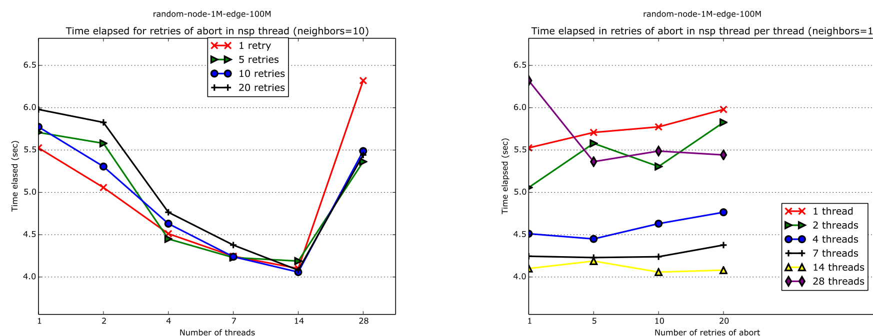
μπορεί να αποθηκεύσει περισσότερα του ενός στοιχεία και η συνολική μνήμη αποθηκεύει περισσότερα στοιχεία από πριν.

4.4.2 Αξιολόγηση του παράλληλου αλγορίθμου

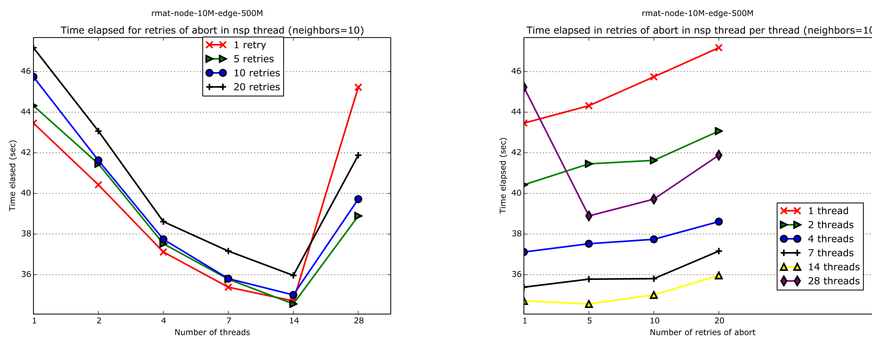
Αξιολόγηση του αριθμού των επαναλήψεων μιας δοσοληψίας πριν την απόκτηση του καθολικού κλειδώματος

Όπως εξηγήθηκε στο προηγούμενο κεφάλαιο, όταν μία δοσηληψία αποτυγχάνει, ξαναπροσπαθεί. Ωστόσο, υπάρχουν δοσοληψίες που θα αποτυγχάνουν συνεχώς. Για παράδειγμα, όταν το read/write σετ μίας δοσοληψίας υπερβεί το read/write σετ του HTM συστήματος, η δοσοληψία θα γίνεται συνέχεια abort εξαιτίας της χωρητικότητας. Σε αυτήν την περίπτωση ένα εναλλακτικό μονοπάτι (fallback path) είναι απαραίτητο. Στο σχήμα μας, το fallback path υλοποιείται με ένα coarse-grained locking κώδικα που χρησιμοποιεί ένα καθολικό κλείδωμα μοιραζόμενο για όλα τα νήματα που προστατεύει το κρίσιμο τμήμα. Αν ένα νήμα αποκτήσει το καθολικό κλείδωμα, κάθε άλλο νήμα δε μπορεί να προχωρήσει στο δικό του κρίσιμο τμήμα και θα γίνεται συνεχώς abort μέχρι να απελευθερωθεί το κλείδωμα.

Αξιολογήσαμε την απόδοση του αλγορίθμου για διαφορετικούς αριθμούς επαναλήψεων μιας δοσοληψίας πριν την απόκτηση του καθολικού κλειδώματος. Πρώτα, το main thread εκτελεί ένα συγκεκριμένο αριθμό επαναλήψεων για μία δοσοληψία και αν ξεπεράσει αυτόν τον αριθμό εξαιτίας διαδοχικών transactional aborts, παίρνει το κλείδωμα, όλα τα helper threads γίνονται aborted και εκτελεί το κρίσιμο τμήμα με ένα coarse-grained locking τρόπο. Τα helper threads δε μπορούν να αποκτήσουν ποτέ το καθολικό κλείδωμα (απεριόριστος αριθμός επαναλήψεων μιας δοσοληψίας), γιατί διαφορετικά θα καθυστερούσαν το main thread και θα παρεμπόδιζαν την πρόοδό του μειώνοντας την απόδοση του αλγορίθμου. Τα σχήματα 4.4 και 4.5 παρουσιάζουν μία αξιολόγηση της απόδοσης για διαφορετικό αριθμό επαναλήψεων μιας δοσοληψίας του main thread (ένας random-node-1M-edge-10M γράφο και ένας rmat-node-10M-edge-500M) εξερευνώντας 5 γείτονες για relaxation σε κάθε δοσοληψία.



Σχήμα 4.4: Ο χρόνος εκτέλεσης για έναν random node-1M-edge-10M γράφο για διαφορετικό αριθμό επαναλήψεων ανά δοσοληψία του main thread.



Σχήμα 4.5: Ο χρόνος εκτέλεσης για έναν rmat node-10M-edge-500M γράφο για διαφορετικό αριθμό επαναλήψεων ανά δοσοληψία του main thread.

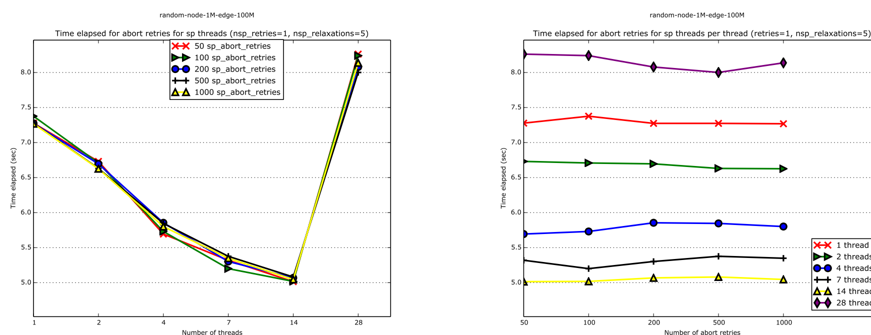
Σύμφωνα με αυτά τα σχήματα, όσο περισσότερες φορές μία δοσοληψία επαναλαμβάνεται, τόσο περισσότερο χρόνο διαρκεί η εκτέλεση του αλγορίθμου. Στην περίπτωση μιας δυνατής επανάληψης ανά δοσοληψία, όταν το main thread αποτύχει να κάνει commit, καταλαμβάνει αμέσως το κλείδωμα και εκτελεί το κρίσιμο τμήμα επιτυχώς, ενώ τα helper threads περιμένουν. Έτσι, η εκτέλεση του main thread είναι πολύ κοντά στη σειριακή εκτέλεση. Αφού το main thread μπορεί να επαναλάβει τη δοσοληψία του περισσότερες φορές (ο αριθμός των επαναλήψεων ανά δοσοληψία αυξάνει), κάθε δοσοληψία διαρκεί περισσότερο όταν διαδοχικά transactional aborts εμφανίζονται. Έτσι, ο χρόνος εκτέλεσης είναι καλύτερος στην περίπτωση μίας δυνατής επανάληψης ανά δοσοληψία. Όσο περισσότερες δυνατές επαναλήψεις ανα δοσοληψία επιτρέπονται τόσο περισσότερο η εκτέλεση του main thread αποκλίνει από τη σειριακή στην περίπτωση πολλών διαδοχικών transactional aborts.

Δεύτερον, παρατηρούμε ότι οι εκτελέσεις με περισσότερες από μία επαναλήψεις ανά δοσοληψία έχουν καλύτερη κλιμακωσιμότητα. Για παράδειγμα, ο χρόνος εκτέλεσης για 14 νήματα στις εκτελέσεις με 5 και 10 δυνατές επαναλήψεις ανά δοσοληψία γίνεται περίπου ίδιος με την περίπτωση μίας δυνατής επανάληψης ανά δοσοληψία (σχήμα 4.5), ενώ στην περίπτωση του ενός νήματος αυτές διαφέρουν σημαντικά. Αυτό συμβαίνει γιατί στις εκτελέσεις με μία δυνατή επανάληψη ανά δοσοληψία τα helper threads αναστέλλονται αμέσως χωρίς να εκτελούν πολλά relaxations. Στις εκτελέσεις με περισσότερες δυνατές επαναλήψεις ανά δοσοληψία τα helper threads έχουν περισσότερο χρόνο να εκτελέσουν relaxations και να τα κάνουν commit, έτσι ώστε το main thread να έχει μεγαλύτερο κέρδος.

Τέλος, στις εκτελέσεις με 28 νήματα εμφανίζεται το NUMA effect της αρχιτεκτονικής που χρησιμοποιήθηκε. Η μεταφορά μιας cache line από τη μνήμη του ενός socket στη μνήμη του άλλου έχει μεγάλο κόστος και επηρεάζει αρνητικά την κλιμακωσιμότητα. Ο χρόνος εκτέλεσης αυξάνεται εξαιτίας των ακριβών μεταφορών cache lines από τη μνήμη ενός άλλου socket. Μπορούμε επίσης να παρατηρήσουμε ότι ο χρόνος εκτέλεσης με μία δυνατή επανάληψη ανά δοσοληψία είναι χειρότερος απο τις άλλες εκτελέσεις με περισσότερες δυνατές επαναλήψεις ανά δοσοληψία. Αυτό οφείλεται στο κοινό μοιραζόμενο καθολικό κλείδωμα. Το main thread γράφει σε αυτό το καθολικό κλείδωμα και κάθε φορά που ένα helper thread ξεκινάει μία δοσοληψία και προσπαθεί να διαβάσει το καθολικό

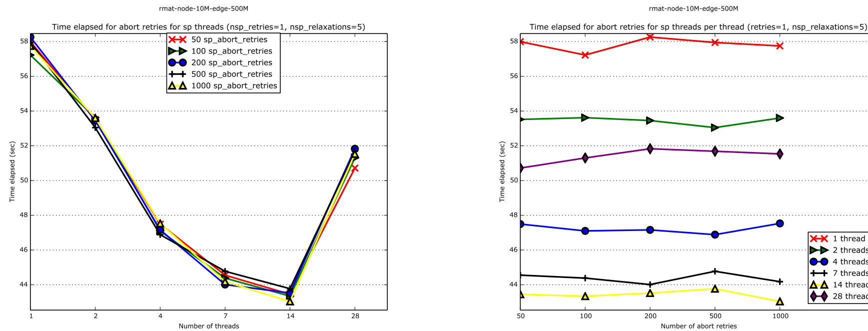
κλείδωμα, πρέπει να το μεταφέρει πιθανώς από μία πιο απομακρυσμένη μνήμη. Αυτό είναι αρκετά δαπανηρό στις εκτελέσεις που το main thread γράφει το καθολικό κλείδωμα πολύ συχνά οπώς στην εκτέλεση με μία δυνατή επανάληψη ανά δοσοληψία. Σε αυτήν την εκτέλεση όταν ένα helper thread, το οποίο βρίσκεται σε διαφορετικό socket από το main thread, διαβάζει το καθολικό κλείδωμα, πρέπει να το μεταφέρει από μία απομακρυσμένη μνήμη γιατί έχει γραφτεί από το main thread με μεγάλη πιθανότητα. Από την άλλη, σε εκτελέσεις με περισσότερες δυνατές επαναλήψεις ανά δοσοληψία το main thread δε γράφει τόσο συχνά το καθολικό κλείδωμα και έτσι, τα helper threads δεν εκτελούν δαπανηρές μεταφορές του καθολικού κλειδώματος σε κάθε δοσοληψία τους. Ωστόσο, κάθε δοσοληψία διαρκεί περισσότερο στην περίπτωση διαδοχικών transactional aborts.

Ακολούθως, εξετάσαμε τον αριθμό των δυνατών επαναλήψεων ανά δοσοληψία στα helper threads. Στις προηγούμενες εκτελέσεις τα helper threads μπορούσαν να επαναλάβουν μία δοσοληψία μέχρι να κάνουν commit ή να γίνουν ρητά abort από το main thread όταν η μεταβλητή "done" γίνει 1. Προσπαθήσαμε να περιορίσουμε τον αριθμό των δυνατών επαναληψεων ανά δοσοληψία στα helper threads, έτσι ώστε να μειωθούν τα conflict aborts με το main thread. Τα helper threads μπορούν να κάνουν ένα συγκεκριμένο αριθμό επαναλήψεων ανά δοσοληψία και αν ξεπεράσουν αυτόν τον αριθμό θα περιμένουν σε ένα while loop μέχρι το main thread να τα ειδοποιήσει να προχωρήσουν στην επόμενη επανάληψη του εξωτερικού loop. Υποθέτουμε ότι επαναλαμβάνοντας μία δοσοληψία για να κάνει commit μετά από ένα συγκεκριμένο αριθμό επαναλήψεων μπορεί μόνο να προκαλέσει conflict aborts και δε μπορεί ποτέ να οδηγήσει σε ένα transactional commit. Τα σχήματα 4.6 και 4.7 απεικονίζουν τα αποτελέσματά μας για διαφορετικό αριθμό δυνατών επαναλήψεων ανά δοσοληψία για τα helper threads.



Σχήμα 4.6: Ο χρόνος εκτέλεσης στον random node-1M-edge-10M γράφο για διαφορετικό αριθμό επαναλήψεων ανά δοσοληψία των helper threads.

Τα αποτελέσματά μας δείχνουν ότι αν και περιορίσαμε των αριθμό των δυνατών επαναλήψεων ανά δοσοληψία, ο συνολικός χρόνος εκτέλεσης του αλγορίθμου δε βελτιώθηκε. Τα helper threads μπορούν να κάνουν commit τις δοσοληψίες τους επαναλαμβάνοντας αυτές λιγότερες φορές από το δοσμένο αριθμό των επαναλήψεων. Ακόμα και αν μειώσαμε τον αριθμό των επαναλήψεων σε ένα πολύ μικρό αριθμό, 50 επαναλήψεις, ο αριθμός μοιάζει με άπειρος. Έτσι, συμπεραίνουμε ότι δεν υπάρχει κέρδος περιορίζοντας τον αριθμό



Σχήμα 4.7: Ο χρόνος εκτέλεσης σε ένα rmat node-10M-edge-500M γράφο για διαφορετικό αριθμό δυνατων επαναλήψεων ανά δοσοληψία των helper threads.

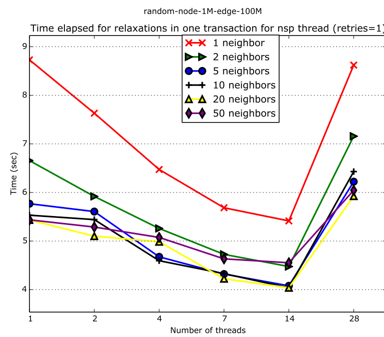
των δυνατών επαναλήψεων ανά δοσοληψία για τα helper threads.

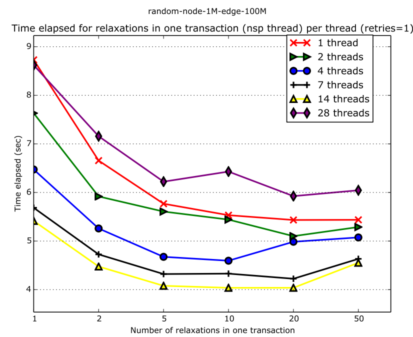
Αξιολόγηση του αριθμού των γειτόνων που εξετάζονται για relaxation ανά δοσοληψία

Για να αποφύγουμε το overhead πολλών μικρών διαδοχικών δοσοληψιών, υλοποιήσαμε ένα πιο coarse-grained σχήμα στις δοσοληψίες για να ελέγχουμε τους γείτονες του τρέχοντος κόμβου και να εκτελούμε relaxations σε αυτούς. Αν ελέγχουμε πάνω από ενα γείτονα σε μία δοσοληψία, το overhead εκτέλεσης πολλών δοσοληψιών μειώνεται, καθώς έχουμε μεγαλύτερες και λιγότερες δοσοληψίες. Ωστόσο, σε μεγάλους γράφους η εκτέλεση μιας πιο coarse-grained δοσοληψίας μπορεί να οδηγήσει σε capacity transactional aborts, αφού αυτό το σχήμα κάνει πρόσβαση σε ένα μεγάλο κομμάτι της μνήμης και μπορεί να υπερβεί το read ή το write set του HTM συστήματος. Έτσι, πρέπει να εξετάσουμε τον αριθμό των γειτόνων που ελέγχονται για relaxation σε μία δοσοληψία.

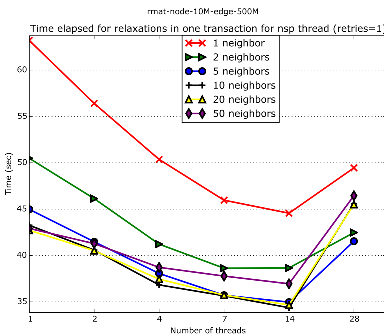
Αξιολογήσαμε τον αλγόριθμο όπως περιγράφεται στα listnings 4.2 και 4.3 για το main και τα helper threads, αντίστοιχα. Πιο συγκεκριμένα, εκτελέσαμε μία coarse-grained δοσοληψία για γείτονες στο main thread, ενώ τα helper threads ελέγχουν μόνο ένα γείτονα για relaxation σε κάθε δοσοληψία. Εκτελέσαμε τον αλγόριθμο για διαφορετικά μεγέθη γράφων για 1, 2, 5, 10, 20 και 50 γείτονες προς εξέταση για relaxation σε μία μόνο δοσοληψία. Τα σχήματα 4.8 και 4.9 δείχνουν τα αποτελέσματά μας για ένα γράφο node-1M-edge-100M graph και ένα rmat node-10M-edge-500M γράφο.

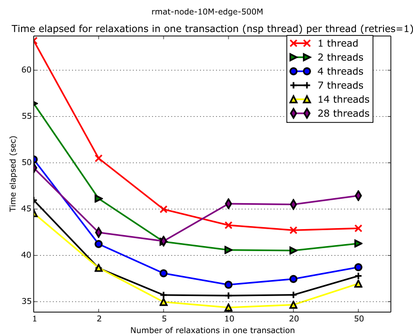
Τα αποτελέσματά μας επιβεβαιώνουν την υπόθεσή μας για μία coarse-grained δοσοληψία. Παρατηρούμε ότι η περίπτωση που ελέγχεται ένας γείτονας ανά δοσοληψία έχει το χειρότερο συνολικό χρόνο, αφού υπάρχουν πολλές μικρές διαδοχικές δοσοληψίας που αδειάζουν και γεμίζουν cache lines της μνήμης και αυτό είναι αρκετά χρονοβόρο. Καθώς ο αριθμός των γειτόνων που ελέγχονται ανά δοσοληψία αυξάνεται ο συνολικός χρόνος βελτιώνεται μέχρι ένα συγκεκριμένο αριθμό γειτόνων. Στους μικρούς γράφους (random-node-1M-edge-100M σχήμα 4.8) ο καλύτερος συνολικός χρόνος εκτέλεσης εμφανίζεται για 20 γείτονες προς εξέταση σε μία δοσοληψία, ενώ στους μεγαλύτερους γράφους (rmat-





Σχήμα 4.8: Ο χρόνος εκτέλεσης για ένα γράφο random node-1M-edge-100M για διαφορετικό αριθμό γειτόνων προς εξέταση για relaxation σε μία δοσοληψία του main thread.

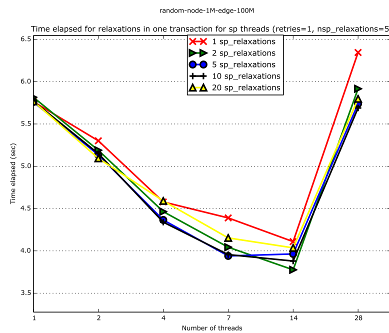


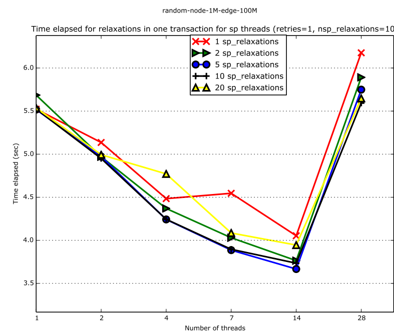


Σχήμα 4.9: Ο χρόνος εκτέλεσης για ένα γράφο rmat node-10M-edge-500M για διαφορετικό αριθμό γειτόνων προς εξέταση για relaxation σε μια δοσοληψία του main thread.

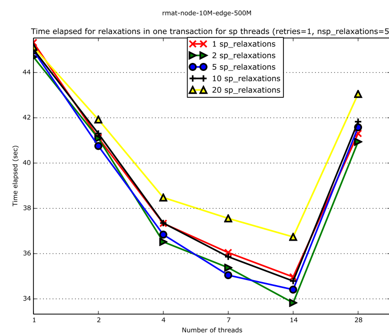
node-10M-edge-500M σχήμα 4.9) ο καλύτερος χρόνος εκτέλεσης εμφανίζεται στην περίπτωση των 10 γειτόνων. Κατά συνέπεια, μπορούμε να συμπεράνουμε ότι όσο μεγαλύτερος είναι ο γράφος, τόσο λιγότερο coarse-grained πρέπει να είναι μία δοσοληψία, αφού οι πιο coarse-grained δοσοληψίες οδηγούν σε capacity transactional aborts στους μεγάλους γράφους.

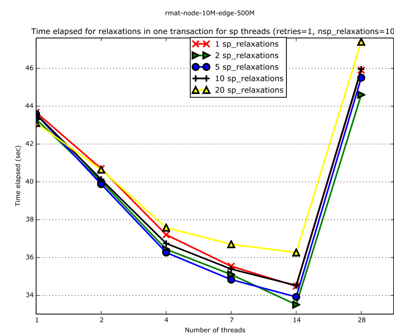
Δεύτερον, η περίπτωση που ελέγχεται ένας γείτονας ανά δοσοληψία κλιμακώνει καλύτερα, καθώς ο αριθμός των νημάτων αυξάνεται. Σε αυτήν την εκτέλεση ο χρόνος για το ένα νήμα είναι ο χειρότερος και ετσι, όσο πιο πολλά νήματα προσθέτουμε, τόσο περισσότερο κέρδος έχουμε. Οι εκτελέσεις με πάνω από έναν γείτονα προς εξέταση ανά δοσοληψία δεν είναι τόσο χρονοβόρες και προσθέτοντας περισσότερα νήματα δεν έχουμε τόσο μεγάλο κέρδος (μικρότερη κλιμακωσιμότητα) από την περίπτωση εξέτασης ενός γείτονα ανά δοσοληψία. Τέλος, οι εκτελέσεις με 28 νήματα είναι χρονοβόρες επειδή η NUMA αρχιτεκτονική του συστήματός μας προκαλεί ακριβές μεταφορές cache lines από το ένα socket στο άλλο.





Σχήμα 4.10: Ο χρόνος εκτέλεσης για ένα γράφο random node-1M-edge-100M για διαφορετικό αριθμό γειτόνων προς εξέταση για relaxation σε μια δοσοληψία των helper threads.





Σχήμα 4.11: Ο χρόνος εκτέλεσης για ένα γράφο rmat node-10M-edge-500M για διαφορετικό αριθμό γειτόνων προς εξέταση για relaxation σε μια δοσοληψία των helper threads.

Στο επόμενο βήμα, εκτελέσαμε την ίδια αξιολόγηση και για τα helper threads, επίσης. Κρατήσαμε σταθερό τον αριθμό των γειτόνων για το main thread σε 5 και έπειτα σε 10 και υλοποιήσαμε επίσης μία coarse-grained δοσοληψία για τα helper threads (1, 2, 5, 10, 20 γείτονες προς εξέταση για relaxation ανά δοσοληψία). Ο σκοπός είναι να αναλύσουμε ποια εξέταση έχει την καλύτερη κλιμακωσιμότητα, καθώς όλες οι εκτελέσεις ξεκινούν από το ίδιο σημείο. Τα αποτελέσματά μας φαίνονται στα σχήματα figures 4.10 και 4.11.

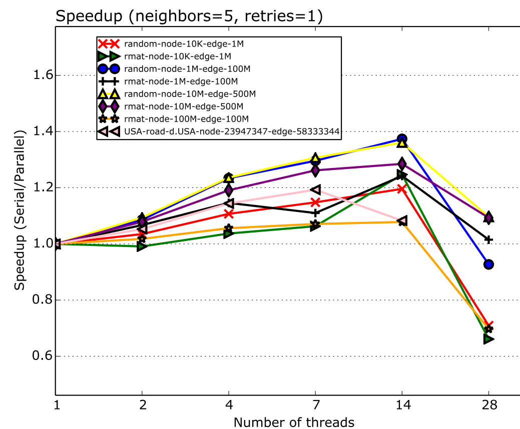
Στο random-node-1M-edge-100M γράφο οι εκτελέσεις με 5 και 10 γείτονες προς εξέταση ανά δοσοληψία έχει την καλύτερη κλιμακωσιμότητα. Όμοια με τις παραπάνω εκτελέσεις, όσο η δοσοληψία γίνεται πιο coarse-grained, τόσο αποφεύγουμε δαπανηρές διαδοχικές μικρές δοσοληψίες, αλλά δεν έχουν κέρδος αν ξεπεράσουμε ένα συγκεκριμένο αριθμό γειτονων. Στο rmat-node-10M-edge-500M γράφο οι εκτελέσεις των 2 και 5 γειτόνων προς εξέταση ανά δοσοληψία δίνουν την καλύτερη κλιμακωσιμότητα. Εφόσον ο γράφος είναι μεγαλύτερος, η δοσοληψία πρέπει να είναι λιγότερο coarse-grained από αυτές σε μικρότερους γράφους. Αυτός κάνει πρόσβαση σε μεγαλύτερο δυαδικό σωρό

και μπορεί να υπερβεί το read/write σετ του HTM συστήματος ελέγχοντας για relaxations μικρότερο αριθμό γειτόνων μέσα σε μία δοσοληψία.

4.5 Αποτελέσματα

4.5.1 Η απόδοση του αλγορίθμου

Στην αξιολόγηση της απόδοσης του αγλορίθμου δοκιμάσαμε γράφους διαφορετικής πυκνότητατας και δομής. Χρησιμοποιήσαμε γράφους με 10K, 1M, 10M και 100M κορυφές τύπου Random και R-MAT. Επίσης, αξιολογήσαμε τον αλγόριθμο για ένα πραγματικό οδικό δίκτυο, ένα οδικό δίκτυο στην Αμερική (USA-road-d.USA). Το σχήμα 4.12 παρουσιάζει την απόδοση που πετύχαμε με την υλοποίηση του αλγοριθμόυ του Dijkstra που περιγράψαμε (listings 4.2 και 4.3). Η απόδοση για n νήματα υπολογίζεται ως ο λόγος του χρόνου εκτέλεσης του σειριακού αλγορίθμου προς την εκτέλεση του αλγορίθμου με n νήματα, n-1 από τα οποία είναι helper threads. Το προτεινόμενο σχήμα απέδωσε σημαντικό κέρδος στις περισσότερες περιπτώσεις. Η μέγιστη απόδοση ήταν 1.39 για τον γράφο random-node-1M-edge-100M με 14 νήματα.



Σχήμα 4.12: Η απόδοση του αλγορίθμου για διαφορετικό πλήθος νημάτων για γράφους διαφορετικής πυκνότητας.

Όπως εξηγείται στο [21] στη σειριακή εκτέλεση, ο χρόνος μπορεί να εκτιμηθεί ως:

$$T_{serial} = n * \mathcal{O}(\log n) + d * n * \mathcal{O}(\log n), \tag{4.1}$$

όπου n αντιπροσωπεύει τον αριθμό των κορυφών στο γράφο και d τη μέση τιμή εξερχόμενων ακμών στις κορυφές. Η ExtractMin() συνάρτηση ξοδεύει χρόνο $n * \mathcal{O}(\log n)$ και η DecreaseKey $d * n * \mathcal{O}(\log n)$, περίπου.

Ο χρόνος εκτέλεσης του παράλληλου σχήματος που περιγράφηκε μπορεί να εκτιμηθεί ως:

$$Tparallel = n * \mathcal{O}(\log n) + a * d * n * \mathcal{O}(\log n), a < 1 \quad (4.2)$$

όπου a είναι ο λόγος των DecreaseKey() λειτουργιών του main thread προς αυτές που εκτελέστηκαν στη σειριακή περίπτωση. Μία προσέγγιση της απόδοσης με βάση τα relaxations του main thread που όμως δε λαμβάνει υπόψιν το χρόνο που ξοδεύεται στο συγχρονισμό των νημάτων και τις καθυστερήσεις από διαδοχικά transactional aborts μπορεί να υπολογιστεί ως εξής:

$$s = \frac{1 + d}{1 + a * d} \quad (4.3)$$

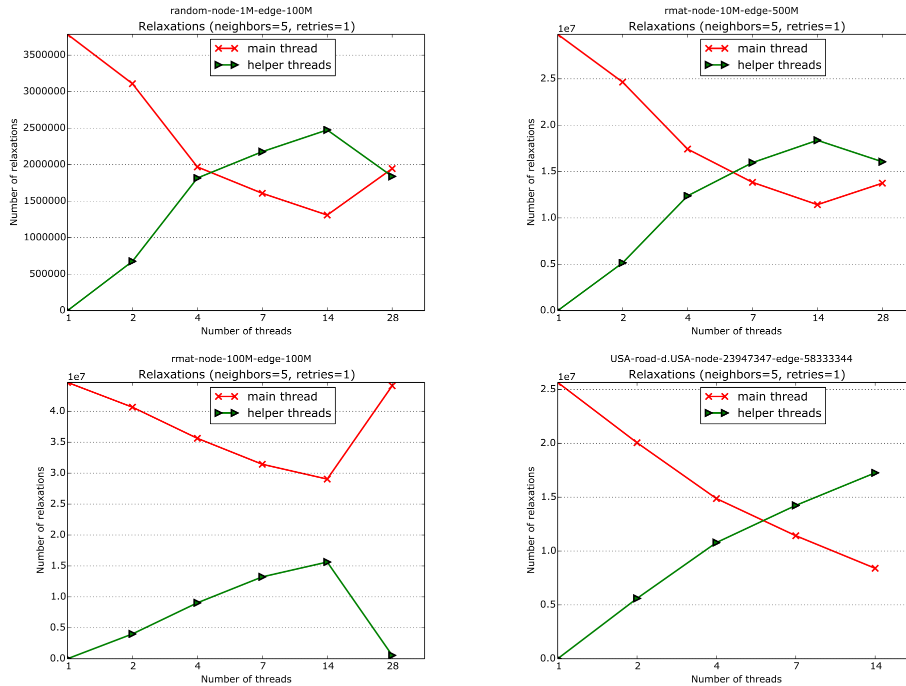
Πρέπει να παρατηρήσουμε ότι η απόδοση συνδέεται με την πυκνότητα του γράφου. Το συμπέρασμα αυτό μπορεί να εξαχθεί και από τον ορισμό της απόδοσης 4.3. Σύμφωνα με τα αποτελέσματα του σχήματος 4.12, για τους πιο πυκνούς γράφους, η απόδοση είναι μεγαλύτερη, καθώς μπορεί να εξαχθεί περισσότερος παραλληλισμός από τον εσωτερικό βρόχο του αλγορίθμου. Αντιθέτως, οι αραιοί γράφοι όπως ο rmat-node-100M-edge-100M γράφος, επιτρέπουν περιορισμένο χώρο για παραλληλισμό και οδηγούν σε χαμηλή απόδοση. Επιπλέον, το σχήμα αυτό αποκαλύπτει ότι η απόδοση αυξάνεται όσο χρησιμοποιούμε περισσότερα νήματα. Η απόδοση βελτιώνεται μέχρι ένα μέγιστο σημείο και μετά από το οποίο η χρήση περισσοτέρων νημάτων οδηγεί σε υποβάθμιση της απόδοσης. Ο αριθμός των νημάτων για να επιτευχθεί αύτο το μέγιστο σημείο σχετίζεται με την πυκνότητα του γράφου. Για παράδειγμα, στον αραιό γράφο rmat-node-100M-edge-100M η αύξηση των νημάτων από 7 σε 14 μειώνει ελάχιστα την απόδοση και στο γράφο USA-road-d.USA-node-23947347-edge-58333344 η απόδοση παραμένει σχεδόν η ίδια. Τέλος, στην περίπτωση των 28 νημάτων η απόδοση σε όλους τους γράφους υποβαθμίζεται εξαιτίας του NUMA effect. Το σύστημά μας είναι μία NUMA αρχιτεκτονική με 14 νήματα σε κάθε socket. Έτσι, η χρήση 28 νημάτων σε δύο sockets οδηγεί σε ακριβές μεταφορές cache line από το ένα socket στο άλλο και επηρεάζει αρνητικά την κλιμακωσιμότητα.

4.5.2 Αναλύοντας περισσότερο τα αποτελέσματα

Σε αυτήν την ενότητα θα προσπαθήσουμε να αναλύσουμε περισσότερο το σχήμα που περιγράψαμε. Εξετάσαμε το κέρδος του main thread σε relaxation, το ποσοστό των abort και το χρόνο που ξοδεύεται σε κάθε λειτουργία του main thread και χρησιμοποιήσαμε όλους τους προηγούμενους γράφους. Παρουσιάζουμε κάποιες αντιπροσωπευτικές εκτελέσεις με γράφους διαφορετικής πυκνότητας, καθώς οι υπόλοιποι έχουν παρόμοια συμπεριφορά.

Το σχήμα 4.13 δείχνει την κατανομή των relaxations που εκτελέστηκαν ανάμεσα στο main και τα helper threads. Όσο ο αριθμός των νημάτων αυξάνεται, τόσο τα relaxations που κάνει το main thread μειώνονται και αυτά των helper threads αυξάνονται, δικαιολογώντας την βελτίωση στην απόδοση. Στον αραιό γράφο, τα relaxations των helper threads δε μπορούν ποτέ να ξεπεράσουν αυτά του main thread, καθώς αυτός είναι ένας αραιός

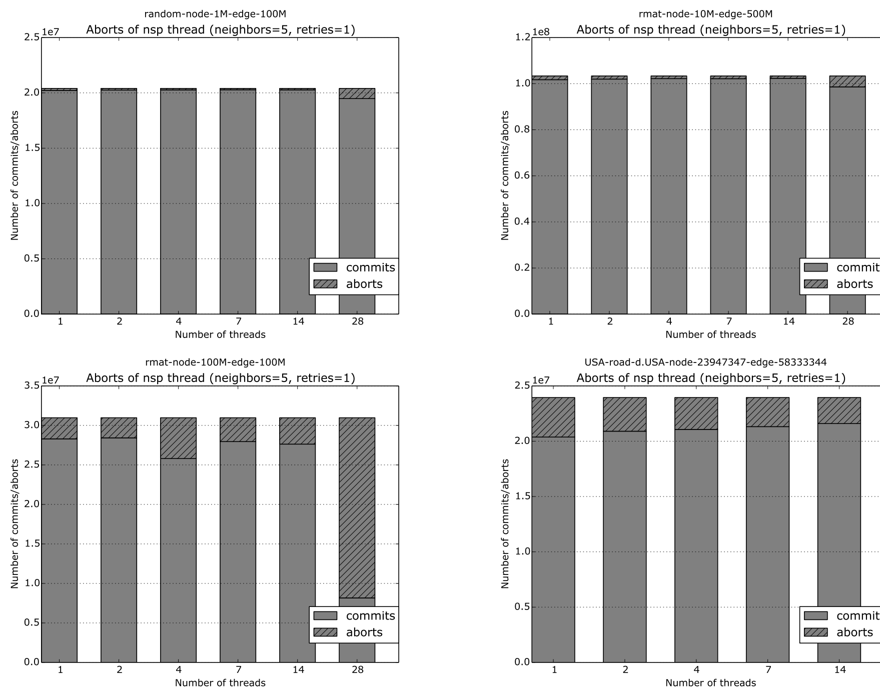
γράφος και τα helper threads δε μπορούν να ξεφορτώσουν πολλά relaxations από το main thread. Χρησιμοποιώντας τα 28 νήματα η κλιμσκωσιμότητα υπβαθμίζεται εξαιτίας του NUMA effect.



Σχήμα 4.13: Κατανομή των relaxations ανάμεσα στο main και τα helper threads.

Το σχήμα 4.14 απεικόνίζει τον αριθμό των commit και abort του main thread. Το main thread υφίσταται ένα πολύ μικρό αριθμό abort, ειδικά στους πυκνούς γράφους. Αυτό σημαίνει ότι ακόμα και όταν τα helper threads δε συμβάλλουν σε χρήσιμη δουλεια, δεν παρεμποδίζουν την πρόοδο του main thread. Το main thread τρέχει σχεδόν στην ταχύτητα της σειριακής εκτέλεσης. Μία σημαντική παρατήρηση επίσης είναι ότι ο αριθμός των transactional aborts στο main thread εξαρτάται από το μέγεθος του write set της δοσοληψίας. Όσο πιο μεγάλο είναι αυτό το σετ, τόσο μεγαλύτερη πιθανότητα για conflict υπάρχει. Επίσης, στην πιο coarse-grained δοσοληψία που υλοποιήσαμε για το main thread, υπάρχει μεγάλη πιθανότητα για capacity aborts στους μεγάλους γράφους σαν το USA οδικό δίκτυο, όπου στην εκτέλεση του main thread υπάρχουν πολλά transactional capacity aborts. Τέλος, η προσθήκη περισσοτέρων νημάτων δεν οδηγεί σε αύξηση του αριθμού των aborts. Έτσι, υποθέτουμε ότι αν δεν υπήρχε το NUMA effect, ο αλγόριθμος θα οδηγούσε σε καλύτερη απόδοση και για περισσότερα από 14 νήματα.

Στην περίπτωση των 28 νημάτων τα transactional aborts αυξάνονται. Μία μόνο δοσοληψία διαρκεί περισσότερο χρόνο εξαιτίας του NUMA effect. Οι μεταφορές των cache line είναι ακριβές και διαρκούν πολύ χρόνο, καθώς αυτές μεταφέρονται από μία απομακρυσμένη μνήμη. Εφόσον η δοσοληψία είνα πιο χρονοβόρα, μπορεί να γίνει abort με μεγαλύτερη πιθανότητα. Πρώτον, data conflicts είναι περισσότερο πιθανό να ανιχνευθούν

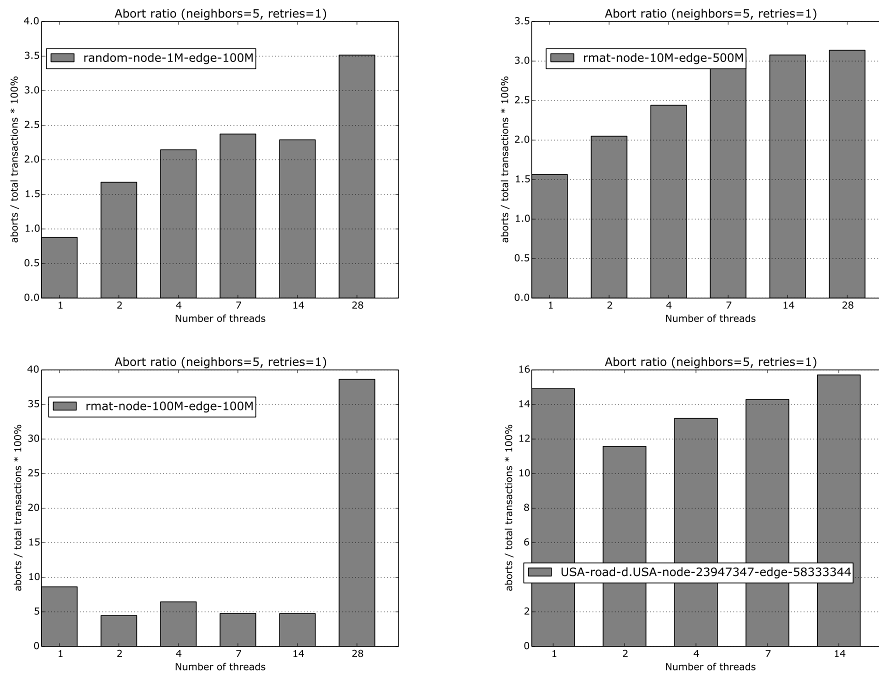


Σχήμα 4.14: Ο αριθμός των commits/aborts του main thread.

σε πιο χρονοβόρες δοσοληψίες. Και δεύτερον, αν μία δοσοληψία διαρκεί περισσότερο από ένα κβάντο χρόνου, ο χρονοδρομολογητής του λειτουργικού συστήματος θα βγάλει τη διεργασία από τον επεξεργαστή και η δοσοληψία θα γίνει abort λόγω time interrupt. Στο rmat-node-100M-edge-100M γράφο, τα transactional aborts στα 28 νήματα εξαιτίας του NUMA effect αυξάνονται σημαντικά, γιατί αυτός είναι ο μεγαλύτερος γράφος και μία δοσοληψία διαρκεί πάρα πολύ χρόνο.

Το σχήμα 4.15 παρουσιάζει το ποσοστό των συνολικών transactional aborts για όλα τα νήματα στο συνολικό αριθμό δοσοληψιών. Ομοίως, οι γράφοι με υψηλη πυκνότητα έχουν μικρό ποσοστό transactional aborts, κάτι που δικαιολογεί και την απόδοση που έχουν, ε-νώ οι αραιοί γράφοι έχουν υψηλότερο ποσοστό συνολικών aborts. Το μικρό ποσοστό των transactional aborts δείχνει ότι οι περισσότερες προσβάσεις στην κοινή δομή δεδομένων είναι non-conflicting. Επιπλέον, παρατηρούμε ότι στους πυκνούς γράφους, καθώς αριθ-μός των νημάτων αυξάνεται, το ποσοστό των transactional aborts αυξάνεται επίσης, αφού η πιθανότητα να υπαρξουν περισσότερες conflicting προσβάσεις είναι μεγαλύτερη. Αντι-θέτως, στους αραιούς γράφους όπως στον rmat-node-100M-edge-100M γράφο, ο αριθμός των transactional aborts δεν αυξάνεται καθώς προσθέτουμε περισσότερα νήματα επειδή η πιθανότητα για conflicting προσβάσεις είναι μικρή στους αραιούς γράφους. Ομοίως με προηγούμενα σχήματα, το ποσοστό το transactional aborts αυξάνεται στην περίπτωση των 28 νημάτων, αφού η δοσοληψία διαρκεί περισσότερο εξαιτίας του NUMA effect και γίνεται περισσότερο επιρρεπής σε transactional abort.

Το σχήμα 4.16 δείχνει το χρόνο που ξοδεύεται από το main thread στην ExtractMin()

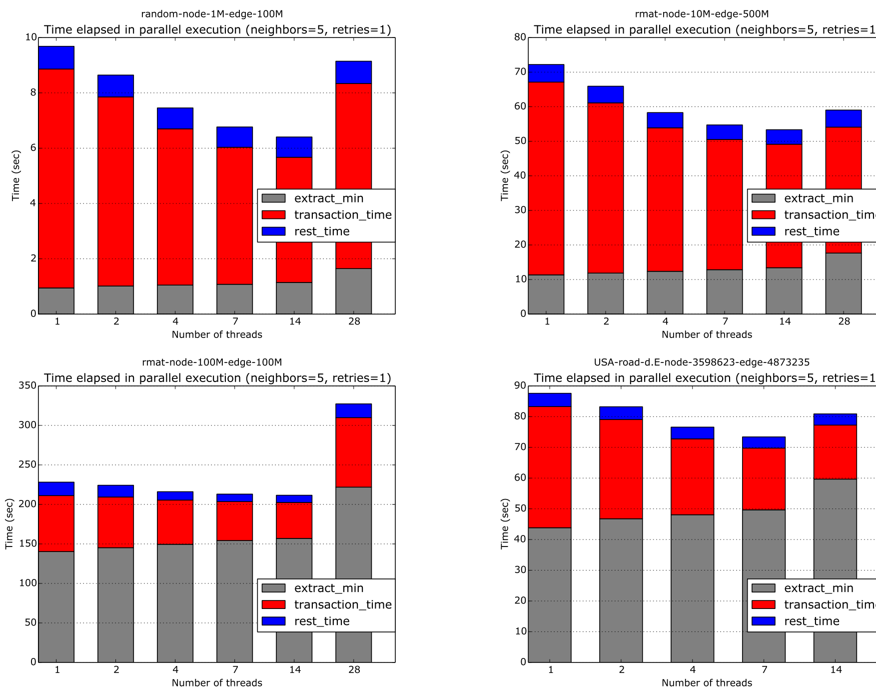


Σχήμα 4.15: Το ποσοστό των συνολικών transactional aborts για όλα τα νήματα στο συνολικό αριθμό δοσοληψιών.

συνάρτηση, στο transactional μέρος του κώδικα (γραμμές 21-37 στο listing 4.2) και το χρόνο που διαρκούν οι υπόλοιποι υπολογισμοί του main thread. Η ExtractMin() συνάρτηση παραμένει σταθερή για κάθε γράφο, καθώς δεν επηρεάζεται στο σχήμα που περιγράψαμε. Ο χρόνος που ξοδεύεται σε αυτην αυξάνεται μόνο στην περίπτωση των 28 νημάτων εξαιτίας των ακριβών μεταφορών cache lines στην NUMA αρχιτεκτονική μας. Δεύτερον, η προσθήκη περισσότέρων helper threads μειώνει το χρόνο που ξοδεύεται σε δοσοληψίες, το παράλληλο κομμάτι στο σχήμα μας. Το main thread εκτελεί λιγότερα relaxations. Εκτελεί λιγότερες φορές τη δαπανηρή συνάρτηση DecreaseKey() και παίρνει χρόνο ανάλογο με $\mathcal{O}(\log n)$ (όπου n είναι ο αριθμός των κορυφών του γράφου) και ως αποτέλεσμα ο χρόνος εκτέλεσης σε transactional mode μειώνεται.

4.5.3 Αξιολόγηση της τεχνικής structure padding

Όλες οι προηγούμενες αξιολογήσεις αποτελεσμάτων έγιναν χρησιμοποιώντας την τεχνική του padding στις κοινές δομές. Χρησιμοποιήσαμε padding στον distance και predecessor πίνακα, καθώς επίσης και στους node_array και where_in_heap πινακα, που χρησιμοποιούνται για την αναπαράσταση του γράφου και είναι επίσης κοινές δομές ανάμεσα στα νήματα. Υποθέσαμε ότι αν δε χρησιμοποιούσαμε padding, ο αριθμός των transactional aborts θα ήταν υψηλότερος και έτσι, η εκτέλεση σε transactional mode θα διαρκούσε περισσότερο χρόνο. Διαφορετικά νήματα που θα εκτελούν λειτουργίες σε ανεξάρτητα δεδο-

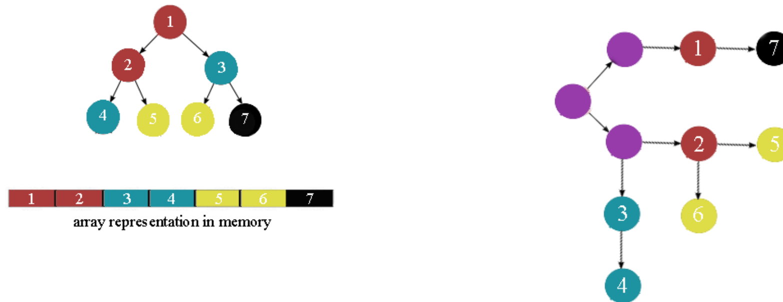


Σχήμα 4.16: Κατανομή του χρόνου στις διάφορες φάσεις εκτέλεσης του main thread.

μένα που βρίσκονται στην ίδια cache line, θα γινόταν abort, καθώς το πραγματικό HTM σύστημά μας ανιχνεύει conflicts σε επίπεδο cache line. Ωστόσο, χωρίς τη χρήση padding η μνήμη μπορεί να αποθηκεύσει περισσότερα στοιχεία της δομής και κατά συνέπεια ένα νήμα μπορεί να βρει ένα στοιχείο στην cache μνήμη με μεγαλύτερη πιθανότητα, να αποφύγει μία ακριβή μεταφορά cache line και να εκμεταλλευτεί τη χρονική και χωρική τοπικότητα.

Με σκοπό να επιβεβαιώσουμε την υποθεσή μας για πιο χρονοβόρες εκτελέσεις στην περίπτωση που δε χρησιμοποιούσαμε padding, εκτελέσαμε τον αλγόριθμο αφαιρώντας το padding από τις κοινές δομές δεδομένων. Τα αποτελέσματά μας απεικονίζονται στο σχήμα 4.18. Αυτά αποδεικνύουν ότι η υπόθεσή μας ήταν λάθος, καθώς ο χρόνος που ξοδεύεται σε transactional mode μειώθηκε. Διαφορετικά νήματα κάνουν πρόσβαση σε ανεξάρτητα δεδομένα που βρίσκονται στην ίδια cache line με πολύ μικρή πιθανότητα. Αυτό εξαρτάται από τη μορφή του γράφου. Στην πιο συνηθισμένη περίπτωση, οι ακμές συνδέουν κορυφές που είναι αρκετά απομακρυσμένες η μία από την άλλη και ως αποτέλεσμα δε βρίσκονται στην ίδια cache line. Ως εκ τούτου, μπορούμε να καταλήξουμε ότι αν και δε χρησιμοποιήσαμε padding, αυτό δεν επηρέασε το ποσοστό των aborts. Το σχήμα 4.17α□ παρουσιάζει μία αναπαράσταση του δυαδικού σωρού στη μνήμη (οι κορυφές που μοιράζονται την ίδια cache line απεικονίζονται με το ίδο χρώμα) και το σχήμα 4.17β□ απεικονίζει το πραγματικό δίκτυο στο οποίο οι ακμές συνδέουν κορυφές που βρίσκονται σε διαφορετικές cache lines. Στην περίπτωση 3 νημάτων, το πρώτο νήμα θα εξετάσει για

relaxation την κορυφή 7, το δεύτερο τις κορυφές 5, 6 και το τρίτο την 4. Αυτές οι ταυτόχρονες προσβάσεις εκτελούνται σε κορυφές που βρίσκονται σε διαφορετικές cache lines παρά το ότι δε χρησιμοποιήθηκε padding. Οπότε, δε θα εμφανιστούν transactional aborts.



(α) Αναπαράσταση πίνακα στη μνήμη. **(β)** Το πραγματικό δίκτυο.

Σχήμα 4.17: Παρά το ότι δε χρησιμοποιήθηκε padding, ταυτόχρονα relaxations εκτελούνται σε κορυφές που βρίσκονται σε διαφορετικές cache lines.

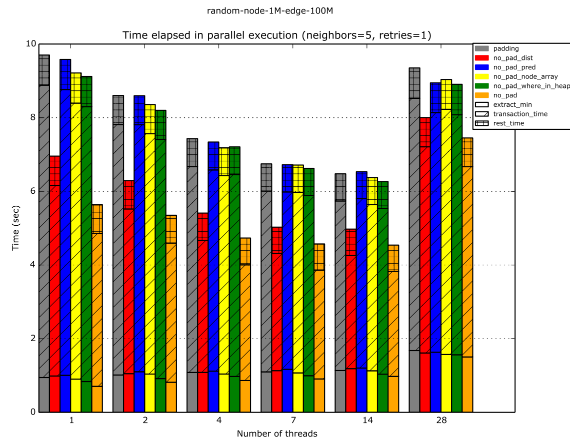
Αντιθέτως, αφαιρώντας το padding ο συνολικός χρόνος εκτέλεσης του αλγορίθμου βελτιώθηκε. Πιο συγκεκριμένα, μειώθηκε ο χρόνος που ξοδεύεται σε transactional mode, ιδιαίτερα όταν αφαιρέθηκε το padding από τον πίνακα distance, γιατί σε αυτόν γίνονται οι περισσότερες προσβάσεις μέσα σε μια δοσοληψία. Το if branch που κάνει πρόσβαση στις υπόλοιπες δομές εκτελείται λιγότερες φορές. Έτσι, το κέρδος από την χρονική και χωρική τοπικότητα σχετίζεται με τον distance πίνακα. Τα νήματα μπορεί να βρουν ένα στοιχείο στη cache μνήμη με μεγαλύτερη πιθανότητα όταν δε χρησιμοποιείται padding και να αποφύγουν μεταφορές cache lines από την κύρια μνήμη. Επιπλέον, αποφεύγοντας συχνές μεταφορές cache lines μπορούμε επίσης να αποφύγουμε καθυστερήσεις στον κοινό δίαυλο μνήμης όταν υπάρχει συμφόρηση μνήμης.

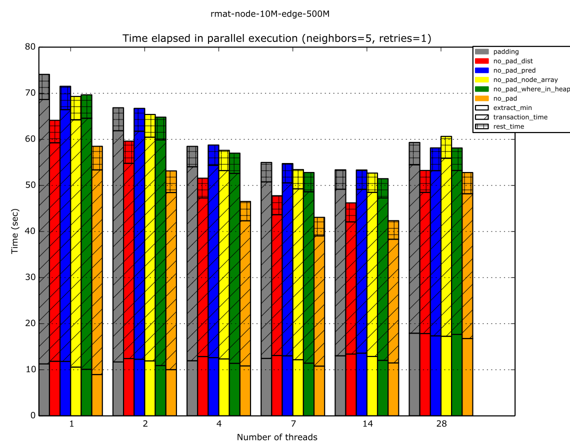
Τέλος, παρατηρούμε ότι παρά το γεγονός ότι αφαιρέσαμε το padding το μοτίβο εκτέλεσης παραμένει το ίδιο. Καθώς ο αριθμός των νημάτων αυξάνεται, η κλιμακωσιμότητα της εκτέλεσης χωρίς τη χρήση padding στις δομές είναι η ίδια με αυτήν με χρήση padding. Η απόδοση του αλγορίθμου έχει τις ίδιες τιμές και η μόνη διαφορά είναι ότι μειώνεται ο χρόνος εκτέλεσης του αλγορίθμου όταν δε χρησιμοποιείται padding. Συνεπώς, η ανάλυσή μας για την κλιμακωσιμότητα και την απόδοση του αλγορίθμου δεν επηρεάζεται.

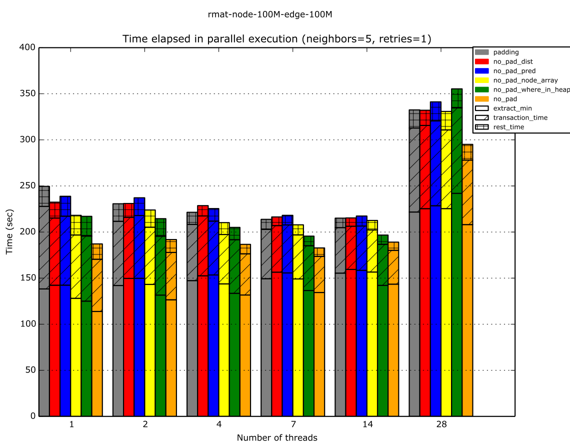
4.6 Χρησιμοποιώντας skip list

4.6.1 Η skip list δομή

Η υλοποίηση που περιγράψαμε χρησιμοποιεί δυαδικό σωρό για την ουρά προτεραιότητας. Σε αυτήν την ενότητα προσπαθήσαμε να αξιολογήσουμε τον αλγόριθμο χρησιμοποιώντας skip list αντί για binary heap. Η DecreaseKey() συνάρτηση παίρνει χρόνο



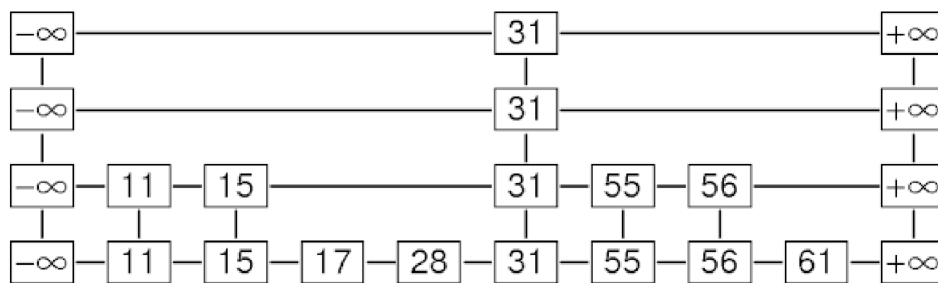




Σχήμα 4.18: Αξιολόγηση της τεχνικής padding στον παράλληλο αλγόριθμο.

ανάλογο του $\mathcal{O}(\log n)$ και για τις δύο δομές δεδομένων και ομοίως και η ReadMin() έχει την ίδια πολυπλοκότητα και για τις δύο δομές. Ωστόσο, η ExtractMin() συνάρτηση παίρνει χρόνο ανάλογο του $\mathcal{O}(1)$ όταν χρησιμοποιούμε skip list, σε αντίθεση με το δυαδικό σωρό όπου παίρνει χρόνο $\mathcal{O}(\log n)$.

Η skip list είναι μία δομή δεδομένων που επιτρέπει γρήγορες αναζητήσεις σε μία ταξινομημένη ακολουθία στοιχείων (key-value pairs). Η γρήγορη αναζήτηση επιτυγχάνεται διατηρώντας μία συνδεδεμένη ιεραρχία υπακολουθιών, κάθε μία προσπερνάει κάποια στοιχεία. Η skip list έχει επίπεδα. Το κατώτερο επίπεδο έιναι μία συνηθισμένη ταξινομημένη λίστα. Ένα κλειδί στο επίπεδο i εμφανίζεται στο επίπεδο i+1 με κάποια σταθερή πιθανότητα p. Έτσι, κάθε στοιχείο της δομής έχει ένα τυχαίο ύψος που αντιπροσωπεύει τα επίπεδα στα οποία εμφανίζεται το κλειδί του στοιχείου. Η skip list έχει ένα μέγιστο ύψος και κάθε φορά που ένα στοιχείο εισάγεται, παίρνει ένα τυχαίο ύψος ανάμεσα σε 1 και στο μέγιστο ύψος της skip list. Το σχήμα 4.19 απεικονίζει ένα παράδειγμα μίας skip list.



Σχήμα 4.19: Ένα παράδειγμα μιας skip list.

Μία αναζήτηση για ένα στόχο-στοιχείο αρχίζει από την κορυφή της λίστας από το υψηλότερο επίπεδο και προχωράει οριζόντια μέχρι να βρεθεί ένα στοιχείο μεγαλύτερο ή ίσο από το κλειδί του στόχου-στοιχείου. Αν το κλειδί του στοιχείου είναι ίσο με αυτό του στόχου, τότε η λειτουργία επιστρέφει επιτυχώς. Διαφορετικά, επαναλαμβάνεται η ίδια διαδικασία για το επόμενο πιο χαμηλό επίπεδο της skip list. Ο συνόλικός χρόνος εκτέλεσης της λειτουργίας αναζήτησης είναι ανάλογος με $\mathcal{O}(\log n)$.

Στη skip list, το ύψος του κάθε στοιχείου (αριθμός των επιπέδων του στοιχείου) είναι ένας τυχαίος αριθμός. Έτσι, είναι πιθανό (με πολύ μικρή πιθανότητα) να παραχθεί μία κακώς ισοζυγισμένη δομή. Ωστόσο, οι skip lists δουλεύουν καλά στην πράξη, και ένα τυχαίο σχήμα είναι πολύ εύκολο να υλοποιηθεί σε σχέση με ένα ντετερμινιστικό σχήμα. Τέλος, οι skip lists είναι χρήσιμες δομές για τον παράλληλο προγραμματισμό, όπου εισαγωγές γίνονται σε διαφορετικά μέρη της skip list ταυτόχρονα χωρίς να απαιτείται καθολικό rebalancing στη δομή δεδομένων.

4.6.2 Σύγκριση με το δυαδικό σωρό

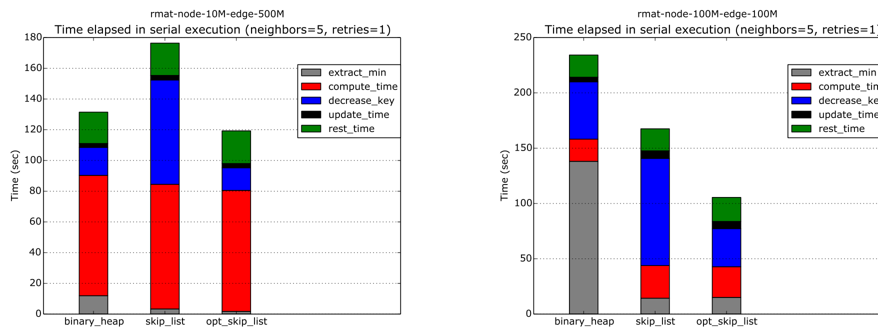
Σε αυτήν την ενότητα θα συγκρίνουμε την υλοποίηση που χρησιμοποιεί skip list για την ουρά προτεραιότητας με την προηγούμενη υλοποίηση. Αρχικά, κατασκευάσαμε μία skip list που είχε ένα στοιχείο για κάθε κορυφή του γράφου. Έτσι, στην DecreaseKey() το στοιχείο που αντιπροσώπευε την εξεταζόμενη κορυφή αφαιρούνταν από τη skip list και τοποθετούνταν σε μία πιο κοντινή θέση στην αρχή της λίστας. Όπως στην περίπτωση του δυαδικού σωρού, αυτή η λειτουργία παίρνει χρόνο ανάλογο με $\mathcal{O}(\log n)$. Ωστόσο, ο χρόνος εκτέλεσης του αλγορίθμου ήταν υπερβολικά μεγαλύτερος και η απόδοση πολύ μικρή. Ο λόγος ήταν ότι η DecreaseKey() συνάρτηση ήταν αρκετά χρονοβόρα.

Στην προσπάθειά μας να βελτιώσουμε αυτήν την υλοποίηση με τη skip list, παρατηρήσαμε ότι η DecreaseKey() εκτελούσε πολλά βήματα για να τοποθετήσει το εξεταζόμενο στοιχείο στη νέα του θέση. Για να τοποθετηθεί ένα στοιχείο στη skip list, χρειάζεται μία διάσχιση από την αρχή της λίστας μέχρι να βρεθεί ένα στοιχείο με ίσο ή μεγαλύτερο κλειδί. Αυτή η διάσχιση έπαιρνε πολλά βήματα, ενώ η τοποθέτηση του στοιχείου στη νέα του θέση στο δυαδικό σωρό απαιτούσε μικρότερο αριθμό swaps των στοιχείων της δομής. Επίσης, παρατηρήσαμε ότι η skip list είχε πάρα πολλά στοιχεία με το ίδιο κλειδί κατά τη διάρκεια της εκτέλεσης του αλγορίθμου. Ενώ έτρεχε ο αλγόριθμος, οι αποστάσεις των κορυφών από τις πηγές ενημερώνοταν με την ίδια τιμή με πολύ μεγάλη πιθανότητα. Κατά συνέπεια, η skip list είχε πολλά στοιχεία με το ίδιο κλειδί, απαιτούσε πολύ μνήμη και η διάσχιση από την κορυφή της λίστας στο επιθυμητό στοιχείο ήταν αρκετά χρονοβόρα εξαιτίας του ότι έπρεπε να προσπεραστούν πολλά στοιχεία με το ίδιο κλειδί.

Έτσι, υλοποιήσαμε μία βελτιστοποιημένη skip list (optimized skip list) που περιείχε μόνο διακριτά κλειδιά. Κάθε στοιχείο της λίστας έχει ένα μοναδικό κλειδί και μία απλή εσωτερική λίστα που αποθηκεύει τις κορυφές που έχουν την ίδια απόσταση-κλειδί από την πηγή. Με αυτόν τον τρόπο, η skip list απαιτεί λιγότερη μνήμη, αφού έχει λιγότερα στοιχεία, και η DecreaseKey() συνάρτηση παίρνει λιγότερα βήματα, αφού διασχίζει μικρότερο αριθμό στοιχείων.

Στην αξιολόγησή μας πρώτα συγκρίναμε τη σειριακή εκτέλεση για τις 3 διαφορετικές δομές, το binary heap, την απλή skip list και την optimized skip list. Το σχήμα 4.20 δείχνει τα αποτελέσματά μας για δύο διαφορετικούς γράφους, τους rmat-node-10M-edge-500M και rmat-node-100M-edge-100M. Η ExtractMin() είναι λιγότερο χρονοβόρα στις υλοποιήσεις που χρησιμοποιούν skip list, αφού παίρνει σταθερό χρόνο ($\mathcal{O}(1)$). Στους μεγάλους γράφους όπως τον rmat-node-100M-edge-100M γράφο, όπου η ExtractMin() αποτελεί μεγάλο ποσοστό του συνολικού χρόνου εκτέλεσης, υπάρχει σημαντικό κέρδος. Επιπλέον, έχουμε να σημειώσουμε ότι η DecreaseKey() παίρνει πολύ χρόνο στην απλή skip list και για τους δύο γράφους. Όπως εξηγήθηκε παραπάνω, αυτό συμβαίνει γιατί χρειάζονται πάρα πολλά βήματα για να τοποθετηθεί το εξαγόμενο στοιχείο στη νέα του θέση, αφού πρέπει να προσπεραστούν πολλά στοιχεία με το ίδιο κλειδί. Στην βελτιστοποιημένη εκδοχή της skip list αποφεύγουμε τέτοιες χρονοβόρες λειτουργίες και τα στοιχεία που πρέπει να προσπεραστούν για να τοποθετηθεί το στοιχείο στη νέα του θέση είναι συγκρίσιμα στο πλήθος με τα swaps στοιχείων που γίνονται στην DecreaseKey() όταν χρησιμοποιείται binary heap.

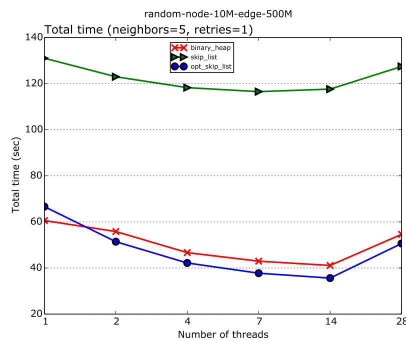
Ακολούθως, αξιολογήσαμε την παράλληλη εκτέλεση του αλγορίθμου για διαφορετι-

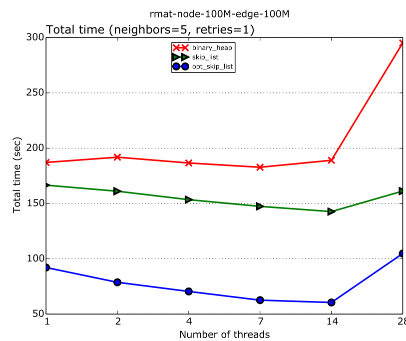


Σχήμα 4.20: Ο χρόνος εκτέλεσης του σειριακού αλγορίθμου για τις 3 διαφορετικές δομές που χρησιμοποιούνται για την ουρά προτεραιότητας.

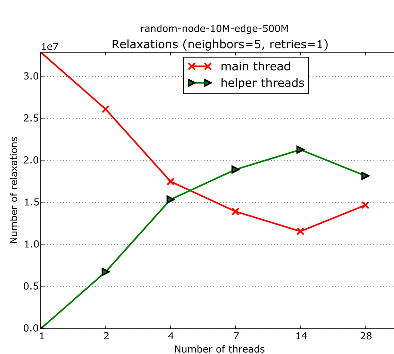
κό αριθμό νημάτων. Το σχήμα 4.21 απεικονίζει το συνολικό χρόνο εκτέλεσης για τους random-node-10M-edge-500M και rmat-node-100M-edge-100M. Όπως στη σειριακή εκτέλεση, η DecreaseKey() συνάρτηση του random-node-10M-edge-500M γράφου στην περίπτωση με την απλή skip list είναι αρκετά χρονοβόρα και έχει το χειρότερο συνολικό χρόνο εκτέλεσης ανάμεσα στις 3 εκτελέσεις. Αυτό συμβαίνει γιατί ο γράφος είναι πυκνός και όσο πιο πυκνός είναι ο γράφος, τόσο περισσότερα relaxations (DecreaseKey()) γίνονται. Έτσι, η DecreaseKey() εκτελείται πολλές φορές και ο συνολικός χρόνος αυξάνεται σημαντικά. Αντιθέτως, στον rmat-node-100M-edge-100M γράφο είναι ένας αραιός γράφος και η DecreaseKey() δεν εκτελείται τόσο πολλές φορές, αφού δεν υπάρχουν πολλές ακμές που να προκαλούν relaxations, οπότε ο συνολικός χρόνος δεν επηρεάζεται. Όπως παρατηρήσαμε στη σειριακή εκτέλεση, η ExtractMin() έχει μεγάλο ποσοστό του χρόνο στην περίπτωση που χρησιμοποιούμε binary heap και κατά συνέπεια η εκτέλεση με το binary heap έχει το χειρότερο συνόλικο χρόνο εκτέλεσης ανάμεσα στις τρεις.

Σχετικά με την κλιμακωσιμότητα της παράλληλης εκτέλεσης παρατηρούμε ότι η εκτέλεση με την βελτιστοποιημένη skip list επιτυγχάνει την υψηλότερη κλιμακωσιμότητα και για τους δύο γράφους. Ωστόσο, στον rmat-node-100M-edge-100M γράφο δεν εμφανίζεται μεγάλη κλιμακωσιμότητα γιατί είναι αραιός γράφος και τα helper threads δεν ξεφορτώνουν πολλές λειτουργίες από το main thread. Η skip list κλιμακώνει καλύτερα γιατί τα helper threads εκτελούν περισσότερα relaxations και το main thread κερδίζει περισσότερο. Όπως παρατηρούμε στα σχήματα 4.22α□ και 4.22β□, ο αριθμός των relaxations του main thread μειώνεται αρκετά περισσότερο όταν χρησιμοποιείται skip list. Από την άλλη πλευρά, η εκτέλεση με το binary heap δε μπορεί να δώσει τόσο μεγάλο κέρδος στα relaxations. Υποθέτουμε ότι τα aborts που γίνονται όταν χρησιμοποιούμε το binary heap είναι abort που θα εκτελούσαν χρήσιμα relaxations. Επιπλέον, η βελτιστοποιημένη skip list κλιμακώνει καλύτερα από την απλή γιατί χρειάζεται μικρότερο read/write set και με αυτό τον τρόπο αποφεύγει δαπανηρές διασχίσεις και capacity transactional aborts. Τέλος, στα 28 νήματα, παρατηρούμε ότι το NUMA effect υποβαθμίζει την κλιμακωσιμότητα όλων των εκτελέσεων εξαιτίας των ακριβών μεταφορών cache lines από το ένα socket στο άλλο.

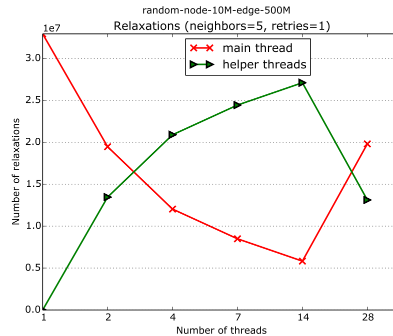




Σχήμα 4.21: Ο χρόνος εκτέλεσης της παράλληλου αλγορίθμου για τις 3 διαφορετικές δομές που χρησιμοποιούνται για την ουρά προτεραιότητας.



(α) Using the binary heap structure.



(β) Using the optimized skip list structure.

Σχήμα 4.22: Η κατανομή των relaxations ανάμεσα στο main και τα helper threads για τον random-node-10M-edge-500M γράφο για 2 διαφορετικές δομές που χρησιμοποιούνται για την ουρά προτεραιότητας.

Στο τελευταίο κομμάτι αυτής της ενότητας παρουσιάζουμε τον αριθμό των transactional commits/aborts του main και των helper thread στις τρεις προηγούμενες εκτελέσεις. Το σχήμα 4.23 απεικονίζει τον αριθμό των commits/aborts του main thread. Ο αριθμός των transactional aborts στην απλή skip list είναι αρκετά υψηλός συγκριτικά με τις άλλες δύο εκτελέσεις, ειδικά στο μεγάλο γράφο. Αυτό οφείλεται στο ότι η διάσχιση στην DecreaseKey() απαιτεί πολύ μνήμη. Απαιτεί πολύ μεγάλο read set και αυτό οδηγεί σε transactional capacity aborts. Επιπλέον, υπάρχει επίσης υψηλή πιθανότητα για data transa-ctional abort. Όσο μεγαλύτερο read set έχει μια δοσοληψία, τόσο μεγαλύτερη πιθανότητα υπάρχει για conflicting accesses. Δεύτερον, παρατηρούμε ότι η εκτέλεση με το binary heap και η εκτέλεση με την βελτιστοποιημένη skip list έχουν συγκρίσιμο αριθμό transactional aborts. Και στις δύο εκτελέσεις το main thread εμφανίζει μικρό αριθμό transactional abort που σημαίνει ότι τα helper threads δεν εμποδίζουν την πρόοδό του.

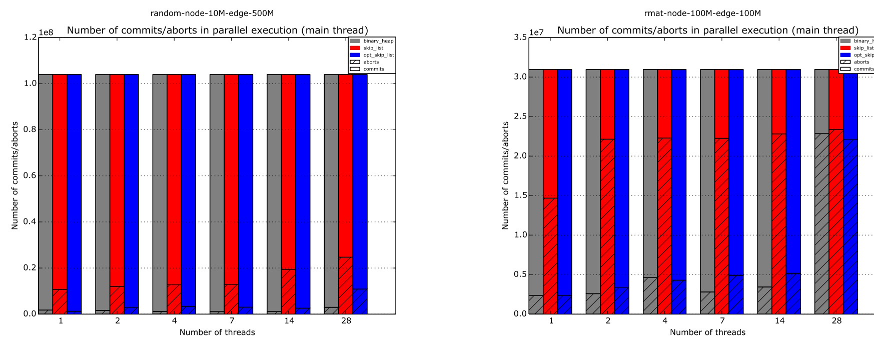
Επιπλέον ο αριθμός των transactional aborts δεν επηρεάζεται συμαντικά από τον αριθμό των νημάτων. Παραμένει σχεδόν ο ίδιος όταν τα νήματα αυξάνονται και καταλήγουμε ότι αν δεν είχαμε το φαινόμενο NUMA στην αρχιτεκοτνική μας, θα μπορούσαμε να έχουμε καλύτερη κλιμακωσιμότητα.

Όπως αναφέραμε και στην ανάλυση με το binary heap το NUMA effect επηρεάζει τα transactional aborts. Εφόσον οι μεταφορές των cache lines είναι ακριβές, η δοσοληψία διαρκεί περισσότερο χρόνο. Κατά συνέπεια, data conflicts είναι πιο πιθανό να ανιχνευθούν σε τέτοιες δοσοληψίες και υπάρχει μεγαλύτερη πιθανότητα για transactional abort λόγω interrupt. Όταν μία δοσοληψία διαρκεί περισσότερο από το κβάντο χρόνου, ο χρονοδρομολογητής του λειτουργικού θα βγάλει τη διεργασία από τον πυρήνα και η δοσοληψία θα γίνει abort.

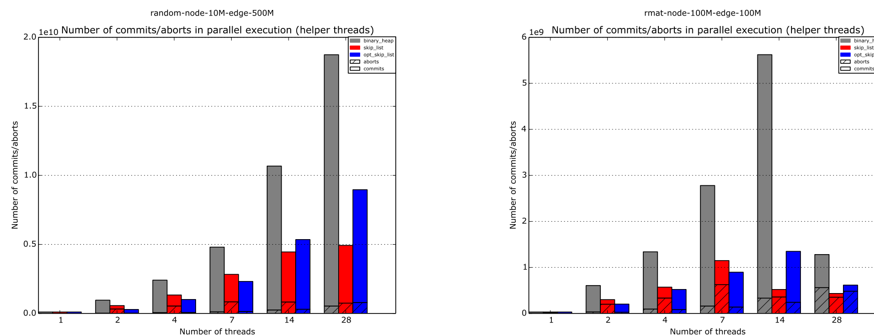
Ομοίως με το main thread, ο αριθμός των transactional aborts για τα helper threads αυξάνεται στην εκτέλεση με την απλή skip list, όπως φαίνεται στο σχήμα 4.24. Η διάσχιση στην DecreaseKey() είναι πολύ δαπανηρή και προκαλούνται πολλά conflicts και για τα helper threads, επίσης. Στις άλλες δύο εκτελέσεις (binary heap και optimized skip list), ο αριθμός των transactional aborts είναι σχετικά χαμηλός και δείχνει ότι οι περισσότερες προσβάσεις στα κοινά δεδομένα είναι non-conflicting. Επιπλέον, ο αριθμός των commits αυξάνεται όταν προσθέτουμε πιο πολλά νήματα, αφού υπάρχουν πιο πολλά νήματα που δρουν. Σε αυτό το σχήμα, ο αριθμός των transactional commits για το binary heap είναι μεγαλύτερος από τις άλλες δύο εκτελέσεις. Αν και συμβαίνει αυτό, η εκτέλεση με το binary heap δεν οδηγεί σε περισσότερα χρήσιμα relaxations. Όπως είδαμε στα σχήματα 4.22α□ και 4.22β□, η εκτέλεση με τη βελτιστοποιημένη skip list κάνει περισσότερα relaxations. Υποθέτουμε ότι στην περίπτωση του binary heap τα helper threads έχουν περισσότερο χρόνο να τρέξουν μέχρι το main thread να τα σταματήσει. Έτσι, ίσως εκτελούν περισσότερες από μία φορές το εξωτερικό while loop κατά την εκτέλεση μιας επανάληψής του από το main thread. Ωστόσο, αφού τα helper threads δεν εξάγουν τα στοιχεία από την ουρά προτεραιότητας, θα διαβάζουν συνεχώς το ίδιο στοιχείο στην readMin() συνάρτηση (Το ν-οστο helper thread διαβάζει το ν-οστο πρώτο στοιχείο της ουράς προτεραιότητας). Μόνο όταν το main thread προχωρήσει στην επόμενη επανάληψή του, τα helper threads διαβάζουν άλλο στοιχείο προς εξέταση. Ως εκ τούτου, στην περίπτωση που τα helper threads έχουν περισσότερο χρόνο να τρέξουν και επαναλαμβάνουν πάνω από μία φορές το εξωτερικο while loop, θα εκτελούν την ίδια διαδικασία (ίδια relaxations) πάνω από μία φορα, εκτελώντας relaxations για το ένα και μοναδικό στοιχείο που μπορούν να διαβάσουν σε κάθε επανάληψη του main thread. Θα εκτελούυν πολλά transactional commits χωρίς να εκτελούν νέα χρήσιμα relaxations.

4.6.3 Αποτελέσματα

Στο τελευταίο μέρος της ανάλυσής μας θα αξιολογήσουμε την απόδοση της παράλληλης εκτέλεσης που χρησιμοποιεί τη βελτιστοποιημένη skip list που υλοποιήσαμε. Χρησιμοποιήσαμε τους ίδιους γράφους με το μέρος της αξιολόγησης της απόδοσης για το binary heap. Το σχήμα 4.25 παρουσιάζει την απόδοση που μπορεί να επιτευχθεί χρησιμοποιώντας τη βελτιστοποιημένη skip list. Το προτεινόμενο σχήμα επιτυγχάνει καλή



Σχήμα 4.23: Ο αριθμός των commits/aborts του main thread στον παράλληλο αλγόριθμο για τις 3 διαφορετικές δομές που χρησιμοποιούνται για την ουρά προτεραιότητας.

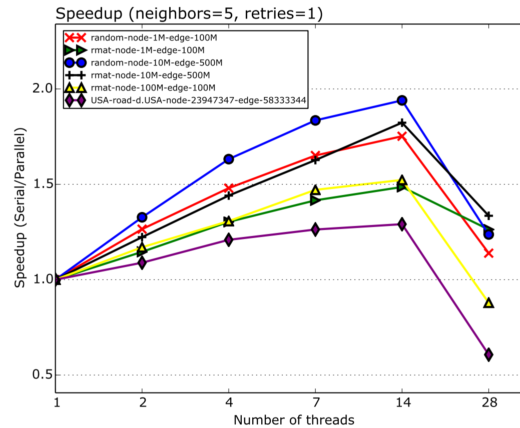


Σχήμα 4.24: Ο αριθμός των commits/aborts των helper threads για τον παράλληλο αλγόριθμο για τις 3 διαφορετικές δομές που χρησιμοποιούνται για την ουρά προτεραιότητας.

απόδοση στους περισσότερους γράφους και η μέγιστη είναι 1.94 για τον πυκνό γράφο random-node-10M-edge-500M (14 νήματα).

Όπως αναφέρθηκε και στην προηγούμενη αξιολόγηση, η απόδοση συνδέεται άμεσα με την πυκνότητα του γράφου. Για πιο πυκνούς γράφους η απόδοση είναι μεγαλύτερη, αφού μπορούμε να εξάγουμε περισσότερο παραλληλισμό. Τα helper threads μπορούν να ξεφορτώσουν περισσότερη εργασία από το main thread σε πυκνούς γράφους, όπου ο αριθμός των ακμών είναι μεγαλύτερος. Αντιθέτως, οι αραιοί γράφοι επιτρέπουν λιγότερο παραλληλισμό. Στην περίπτωση των 28 νημάτων η απόδοση υποβαθμίζεται εξαιτίας του NUMA effect.

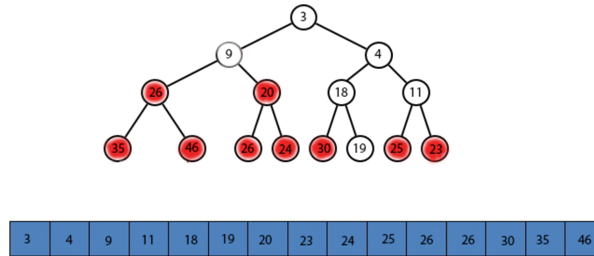
Η υλοποίηση της βελτιστοποιημένης skip list επιτυγχάνει υψηλή απόδοση και κλιμακώνει καλύτερα από την υλοποίηση με το binary heap. Ο κύριος λόγος για αυτό είναι τα relaxations που εκτελούνται από τα helper threads. Όπως ήδη αναφέραμε, χρησιμοποιώντας τη βελτιστοποιημένη skip list τα helper threads εκτελούν πιο πολλά χρήσιμα relaxations. Ο αριθμός τους είναι αρκετά μεγαλύτερος από αυτόν που προκύπτει με τη



Σχήμα 4.25: Η απόδοση του αλγορίθμου για γράφους διαφορετικής πυκνότητας με χρήση της βελτιστοποιημένης skip list.

χρήση του binary heap, όπως φαίνεται στα σχήματα 4.22α και 4.22β, ενώ και οι δύο υλοποιήσεις έχουν συγκρίσιμο αριθμό transactional aborts (σχήματα 4.23 και 4.24). Σε μερικές εκτελέσεις, τα relaxations που εκτελούνται χρησιμοποιώντας τη βελτιστοποιημένη skip list είναι διπλάσιος από αυτά που εκτελούνται όταν χρησιμοποιείται binary heap. Αυτό ίσως οφείλεται στα transactional aborts. Υποθέτουμε ότι στην περίπτωση του binary heap τα transactional aborts που εκτελούνται θα οδηγούσαν σε χρήσιμα relaxations. Εφόσον δε μπορούμε να καθορίσουμε μία πολιτική όταν ανιχνεύεται ένα conflict, δεν είμαστε σίγουροι για το ποια δοσοληψία θα γίνει abort. Είναι πιθανό ότι ένα μεγάλο μέρος των aborted δοσοληψιών στην περίπτωση που χρησιμοποιείται binary heap θα οδηγούσε σε χρήσιμα relaxations, κάτι που δε συμβαίνει στην περίπτωση της βελτιστοποιημένης skip list.

Δεύτερον, το binary heap και η skip list είναι δυο τελείως διαφορετικές δομές δεδομένων. Η skip list είναι μία απόλυτα ταξινομημένη δομή δεδομένων. Έτσι, σε μία δεδομένη χρονική στιγμή, τα helper threads διαβάζουν κορυφές με τις πρώτες ελάχιστες αποστάσεις (κλειδία) από την πηγή και εκτελούν τα relaxations τους. Αντιθέτως, το binary heap δεν είναι μία απόλυτα ταξινομημένη δομή δεδομένων. Σε μία δεδομένη χρονική στιγμή τα helper threads διαβάζουν τα στοιχεία που βρίσκονται υψηλότερα (σε χαμηλό βάθος) στο binary heap, αλλά αυτά τα στοιχεία δεν απαραίτητα οι κορυφές με τις ελάχιστες αποστάσεις από την πηγή. Για παράδειγμα, στο binary heap του σχήματος 4.26 αν είχαμε 5 helper threads, αυτά θα διάβαζαν τα στοιχεία με κλειδιά 9, 4, 26, 20 and 18. Στην περίπτωση που χρησιμοποιούσαμε skip list, τα 5 helper threads θα εξέταζαν τα στοιχεία με κλειδιά 4, 9, 11, 18 και 19, καθώς η skip list είναι απόλυτα ταξινομημένη. Έτσι, καταλήγουμε ότι σε κάθε βήμα του αλγορίθμου τα helper threads εξετάζουν διαφορετικά στοιχεία για να εκτελέσουν τα relaxations τους ανάλογα με τη δομή που χρησιμοποιείται. Υποθέτουμε ότι η καθολική διάταξη που υπάρχει στη skip list, ίσως οδηγεί στο να εξετάζονται κορυφές που έχουν πάρει τη βέλτιστη τιμή τους με μεγαλύτερη πιθανότητα από αυτούς στην



Σχήμα 4.26: Ένα παράδειγμα binary heap. Οι κορυφές με κόκκινο χρώμα δεν έχουν ακόμα αποκτήσει τις βέλτιστες τιμές τους.

περίπτωση που χρησιμοποιείται το binary heap.

Τέλος, για να κάνουμε μία εκτίμηση της αποδόσης χρησιμοποιήσαμε τον τύπο 4.3. Αυτός δίνει μία προσέγγιση της απόδοσης βασισμένη στα relaxations του main thread. Επίσης υπονοείται ότι η απόδοση αυξάνεται με το μέσο όρο εξερχόμενων ακμών. Όσο πιο πυκνός είναι ένας γράφος, τόσος περισσότερος παραλληλισμός μπορεί να εξαχθεί. Ωστόσο, αυτός ο θεωρητικός τύπος είναι μία απλή εκτίμηση και αποτελεί ένα θεωρητικό άνω όριο. Δεν λαμβάνει υπόψιν το χρόνο που ξοδεύεται στο συντονισμό των νημάτων και πιθανές καθυστερήσεις από διαδοχικά transactional aborts. Τα σχήματα 4.27 και 4.28 παρουσιάζουν μία θεωρήτική απόδοση και την απόδοση που πετύχαμε για όλους τους γράφους στην περίπτωση των 14 νημάτων χρησιμοποιώντας τη binary heap δομή και τη βελτιστοποιημένη skip list, αντίστοιχα.

Graph	Ideal Speedup	Speedup achieved
random-node-1M-edge-100M	4	1.45
rmat-node-1M-edge-100M	3.58	1.27
random-node-10M-edge-500M	3.74	1.46
rmat-node-10M-edge-500M	3.09	1.35
rmat-node-100M-edge-100M	1.22	1.1
USA-road-node-23M-edge-58M	1.91	1.08

Σχήμα 4.27: Μια προσέγγιση της απόδοσης που βασίζεται στα relaxations που εκτελεί το main thread στην περίπτωση των 14 νημάτων και η απόδοση που επετεύχθει όταν χρησιμοποιήθηκε binary heap.

Graph	Ideal Speedup	Speedup achieved
random-node-1M-edge-100M	4.57	1.75
rmat-node-1M-edge-100M	5.11	1.49
random-node-10M-edge-500M	5.14	1.94
rmat-node-10M-edge-500M	4.74	1.82
rmat-node-100M-edge-100M	1.79	1.52
USA-road-node-23M-edge-58M	1.48	1.29

Σχήμα 4.28: Μια προσέγγιση της απόδοσης που βασίζεται στα relaxations που εκτελεί το main thread στην περίπτωση των 14 νημάτων και η απόδοση που επετεύχθει όταν χρησιμοποιήθηκε η βελτιστοποιημένη skip list.

Κεφάλαιο 5

Συμπεράσματα και Μελλοντικές Επεκτάσεις

Στο πρώτο μέρος αυτής της διπλωματικής μελετήσαμε παράλληλες δομές δεδομένων, και συγκεκριμένα δυαδικά δέντρα αναζήτησης. Τα δέντρα αναζήτησης είναι η πιο συχνά χρησιμοποιημένη δομή σε εφαρμογές και ο παραλληλισμός τους αποτελεί μία πρόκληση.

Οι πρώτες υλοποιήσεις που παρασουσιάσαμε αποτέλεσαν μία απλοϊκή προσέγγιση δέντρων αναζήτησης. Υλοποιήσαμε δυαδικά δέντρα αναζήτησης, AVL δέντρα και Red-Black δέντρα χρησιμοποιώντας coarse-grained και fine-grained locking ως μηχανισμούς συγχρονισμού μεταξύ των νημάτων. Τα αποτελέσματά μας δείχνουν ότι οι coarse-grained υλοποιήσεις δεν κλιμακώνουν καθώς αυτές δεν παρέχουν παραλληλισμό (σειριακή εκτέλεση) και οι fine-grained υλοποιήσεις κλιμακώνουν για μεγάλα δέντρα μέχρι ένα μικρό αριθμό νημάτων. Τα Red-Black δέντρα έχουν την υψηλότερη απόδοση, αφού είναι δέντρα ισοζυγισμένα ως προς το ύψος.

Από την άλλη πλευρά, οι πιο σύνθετες υλοποιήσεις κλιμακώνουν καλύτερα. Έχουν ένα καλύτερο μηχανισμό συγχρονισμού και κάποιες από αυτές εκτελούν βοηθητικές στρατηγικές όταν εκτελούνται παράλληλα λειτουργίες. Δεύτερον, σύμφωνα με τα αποτελέσματα που παρουσιάστηκαν δεν υπάρχει μεγάλη διαφορά στην απόδοση ανάμεσα στις lock-free και στις lock-based υλοποιήσεις. Ο κύριος στόχος των πιο σύνθετων υλοποιήσεων είναι να μειώσουν τον αριθμό και το granularity των κλειδωμάτων, έτσι ώστε η εκτέλεση να είναι κοντά σε έναν ασύγχρονο αλγόριθμο. Όσο πιο κοντά σε έναν σειριακό αλγόριθμο είναι μία υλοποίηση, τόσο καλύτερα κλιμακώνει. Τέλος, η κλιμακωσιμότητα εξαρτάται από το υλικό. Η NUMA αρχιτεκτονική επηρεάζει την κλιμακωσιμότητα εξαιτίας των ακριβών μεταφορών cache lines.

Εκτός από τις υλοποιήσεις που εξετάστηκαν σε αυτή τη διπλωματική, υπάρχουν αρκετές ενδιαφέρουσες υλοποιήσεις παράλληλων δέντρων αναζήτησης, κάθε μία από τις οποίες έχει το δικό της σύνολο χαρακτηριστικών. Για παράδειγμα, στις δημοσιεύσεις [22], [23], [24], [25] και [26] παρουσιάζονται και άλλα σύνθετα παράλληλα δέντρα αναζήτησης προς μελέτη. Επιπλέον, θα μπορούσαμε να υλοποιήσουμε παράλληλα δέντρα αναζήτησης χρησιμοποιώντας transactional memory σαν μηχανισμό συγχρονισμού. Α-

κόμα και μία απλή coarse-grained locking HTM υλοποίηση από ένα ισοζυγισμένο δέντρο όπως ένα Red-Black δέντρο μπορεί να εμφανίσει υψηλές επιδόσεις. Ωστόσο, για να πετύχουμε κλιμακωσιμότητα σε μεγάλο αριθμό νημάτων, πρέπει ο προγραμματιστής να είναι ενήμερος για τους περιορισμούς που θέτει το HTM σύστημα που χρησιμοποιείται και να προσαρμόσει κατάλληλα τον κώδικα.

Εφόσον ο σκοπός αυτής της ανάλυσης είναι να παρουσιάσει παράλληλες δομές δεδομένων που κλιμακώνουν αποδοτικά και θα μπορούσαν να χρησιμοποιηθούν σε μία παράλληλη εφαρμογή για να βελτιώσουν την απόδοσή της, εκτός από δέντρα αναζήτησης, υπάρχουν πολλές άλλες δομές που θα μπορούσαν να μελετηθούν. Συνδεδεμένες λίστες όπως FIFO queues, πίνακες κατακερματισμού και ουρές προτεραιότητας όπως binary heaps και skip lists είναι επίσης δομές που χρησιμοποιούνται συχνά σε παράλληλες εφαρμογές και θα μπορούσαν επίσης να μελετηθούν.

Στο δεύτερο μέρος αυτής της διπλωματικής, αξιολογήσαμε μία τεχνική παραλληλοποίησης πάνω στον αλγόριθμο του Dijkstra, ο οποίος είναι ένας δύσκολα παραλληλοποιήσιμος αλγόριθμος. Χρησιμοποιήσαμε την υλοποίηση που προτείνεται στα [20], [21] με σκοπό να πετύχουμε υψηλή κλιμακωσιμότητα σε ένα πραγματικό HTM σύστημα.

Ο σκοπός αυτής της ανάλυσης είναι να βρεθεί μία πολιτική που ευνοεί το main thread. Τα helper threads μπορούν να εκτελούν λειτουργίες ταυτόχρονα χωρίς να παρεμποδίσουν την πρόοδο του main thread. Έτσι, για να αποφύγουμε διαδοχικά transactional aborts τα οποία θα καθυστερούσαν την εκτέλεση του main thread, αυτό αποκτά το κλείδωμα άμεσα. Μπορεί να επαναλάβει την εκτέλεση της δουλείας του σε transactional mode μόνο μία φορά. Επιπλέον, τα helper threads δε μπορούν ποτέ να αποκτήσουν το καθολικό κλείδωμα, έτσι ωστε να μην καθυστερήσουν την εκτέλεση του main thread και προσπαθούν συνεχώς να εκτελέσουν το κρίσιμο τμήμα σε transactional mode.

Δεύτερον, υλοποιήσαμε μία πιο coarse-grained δοσοληψία στο main thread για να μειώσουμε το overhead των δοσοληψιών και δοκιμάσαμε το ίδιο και για τα helper threads, επίσης. Ωστόσο, εκτελώντας μία coarse-grained δοσοληψία μπορεί να οδηγήσει σε capacity transactional aborts, ειδικά σε μεγάλους γράφους, εξαιτίας του μεγάλου κομματιού της μνήμης στο οποίο γίνεται πρόσβαση. Επίσης, μελετήσαμε την τεχνική padding στις κοινές δομές δεδομένων και καταλήξαμε ότι το padding δεν επηρεάζει την κλιμακωσιμότητα του αλγορίθμου και παρέχει καλύτερο χρόνο εκτέλεσης, αφού τα νήματα εκμεταλλεύονται τη χρονική και χωρική τοπικότητα.

Ακολούθως, παρουσιάσαμε γράφους για τα relaxations που εκτελούν το main και τα helper threads καθώς επίσης και το ποσοστό των abort στις εκτελέσεις μας. Παρατηρούμε ότι το κέρδος στα relaxations είναι σημαντικό, αλλά δεν είναι αντίστοιχο με το κέρδος σε απόδοση εξαιτίας του κόστους συγχρονισμού των νημάτων. Επιπλέον, τα helper treads ίσως κάνουν relaxations που δεν είναι χρήσιμα καθώς ο αλγόριθμος προχωράει. Σαν μελλοντική επέκταση θα μπορουσε να σχεδιαστεί μία παραλλαγή του αλγορίθμου που παρουσιάσαμε στην οποία θα υπήρχε μικρότερο κομμάτι συγχρονισμού και ο αλγόριθμος θα κλιμάκωνε και σε NUMA αρχιτεκτονικές.

Στο τελευταίο μέρος αυτής της θέσης αξιολογήσαμε τον αλγόριθμο χρησιμοποιώντας skip list για την ουρά προτεραιότητας. Τα αποτελέσματά μας δείχνουν ότι η χρήση μιας απλής skip list δίνει αρκετά χειρότερη απόδοση από το binary heap. Η απλή skip

list απαιτεί πολύ μνήμη και η διάσχιση στα στοιχεία της είναι αρκετά χρονοβόρα. Έτσι, υλοποιήσαμε μία βελτιστοποιημένη skip list που περιέχει μόνο διακριτά κλειδιά. Οι κορυφές που έχουν το ίδιο κλειδί αποθηκεύονται σε μία εσωτερική εμφωλευμένη λίστα του στοιχείου της skip list. Με αυτόν τον τρόπο, η βελτιστοποιημένη skip list χρειάζεται λιγότερη μνήμη, καθώς υπάρχουν πολλοί κόμβοι με το ίδιο κλειδί κατά τη διάρκεια εκτέλεσης του αλγορίθμου και η διάσχιση στα στοιχεία της είναι αρκετά πιο γρήγορη. Αυτή η υλοποίηση δίνει καλύτερη απόδοση από το binary heap και έχει υψηλότερη κλιμακωσιμότητα. Τα helper threads τροποποιούν τοπικά τη skip list και εκτελούν περισσότερα χρήσιμα relaxations από ότι στην περίπτωση που χρησιμοποιείται binary heap.

Η βελτιστοποιημένη skip list δεν περιέχει ένα στοιχείο για κάθε κορυφή του γράφου. Ο αριθμός των στοιχείων εξαρτάται από τον αριθμό των κορυφών που αποκτούν την ίδια απόσταση (κλειδί στη skip list) από την πηγή κατά τη διάρκεια εκτέλεσης του αλγορίθμου. Αυτό εξαρτάται από τα βαρή των ακμών του γράφου. Κατά συνέπεια, δε μπορούμε να συμπεράνουμε ένα σταθερό ύψος που πρέπει να έχει η βελτιστοποιημένη skip list. Πρέπει να εξετάζουμε το μέσο αριθμό των στοιχείων που έχει η skip list κατά τη διάρκεια εκτέλεσης και να θέτουμε ως μέγιστο ύψος το λογάριθμο αυτού του αριθμού. Γενικά, η παράμετρος του μέγιστου ύψους της βελτιστοποιημένης skip list πρέπει να εξετάζεται χωριστά για κάθε γράφο.

Σε ένα δεδομένο βήμα του αλγορίθμου, το πρώτο στοιχείο της βελτιστοποιημένης skip list έχει την ελάχιστη απόσταση-κλειδί από την πηγή. Αυτό το στοιχείο ίσως περιέχει πολλά ids κορυφών του γράφου τα οποία έχουν την ελάχιστη απόσταση από την πηγή σε αυτό το βήμα. Έτσι, μία πρόταση θα ήταν ότι το main thread θα μπορούσε να εξάγει όλο το στοιχείο της skip list και να αναθέτει τις διαφορετικές κορυφές στα helper threads για να εκτελέσουν τα relaxations τους. Με αυτό τον τρόπο, θα είχαμε πολλαπλούς εξερευνημένους κόμβους σε ένα βήμα και αυτό θα οδηγούσε σε μεγαλύτερα κέρδη. Ωστόσο, στην περίπτωση που μία κορυφή εξάγεται από την ουρά προτεραιότητας, πρέπει να εξεταστούν όλες οι ακμές της, διαφορετικά ο αλγόριθμος δε θα είναι ορθός. Ως εκ τούτου, ο αλγόριθμος πρέπει να επανασχεδιασθεί, έτσι ώστε κάθε βήμα να παίρνει χρόνο όσο η πιο χρονοβόρα εξέταση κορυφής ανάμεσα σε όλα τα νήματα. Σε αυτήν την εκτέλεση, το main thread δεν πρέπει να σταματάει τα helper threads όταν αυτά έχουν ακόμα ακμές να εξετάσουν.

Η αξιολόγησή μας εκτελέστηκε στο σύτημα Intel's Haswell HTM. Σαν συνέχεια της δουλειάς που παρουσιάστηκε, ο αλγόριθμος θα μπορούσε να αξιολογηθεί και σε άλλα συστήματα που υποστηρίζουν Hardware Transactional Memory. Θα μπορούσε να εξερευνηθεί η επίδραση από διαφορετικά χαρακτηριστικά των TM συστημάτων, όπως η πολιτική επίλυσης των conflicts, το version management και η πολιτική ανίχνευσης των conflicts. Για παράδειγμα, μία Υβριδική πολιτική επίλυσης συγκρούσεων θα μπορούσε να είναι πιο αποτελεσματική ή ένα σύστημα που έχει μεγαλύτερο read/write transaction set θα ήταν πιο κατάλληλο για μεγαλύτερους γράφους. Ο προγραμματιστής πρέπει να είναι ενημερωμένος για το HTM σύστημα και τα χαρακτηριστικά του και να αλλάζει τον κώδικα κατάλληλα για να επιτευχθεί υψηλή κλιμακωσιμότητα.

Τέλος, στη δημοσίευση [21], τα αποτελέσματα δείχνουν διαφορετικό διαθέσιμο παραλληλισμό κατά τις διαφορετικές φάσεις εκτέλεσης. Καθώς ο αλγόριθμος προχωράει ο

διαθέσιμος παραλληλισμός μειώνεται και τα κέρδη από τη χρήση περισσοτέρων helper thread είναι αμελητέα. Έτσι, σαν μελλοντική επέκταση θα μπορούσαν να εξεταστούν οι διαφορετικές φάσεις του αλγορίθμου και να εξερευνηθούν σχήματα στα οποία ο αριθμός των helper threads και οι λειτουργίες που εκτελούν αυτά θα προσαρμόζονταν δυναμικά.

Βιβλιογραφία

[1] Amdahl, G., *The validity of the single processor approach to achieving large scale computing capabilities*, In Proceedings of AFIPS Spring Joint Computer Conference, Atlantic City, N.J., AFIPS Press, April 1967

[2] Flynn, M., *Some Computer Organizations and Their Effectiveness*, IEEE Transactions on Computers, 1972.

[3] Georgy Adelson-Velsky, G.; Evgenii Landis (1962)., *An algorithm for the organization of information*, Proceedings of the USSR Academy of Sciences (in Russian) 146: 263–266. English translation by Myron J. Ricci in Soviet Math. Doklady, 3:1259–1263, 1962.

[4] Leonidas J. Guibas and Robert Sedgewick (1978)., *A Dichromatic Framework for Balanced Trees*, Proceedings of the 19th Annual Symposium on Foundations of Computer Science. pp. 8–21. doi:10.1109/SFCS.1978.3.

[5] Siakavaras D., Nikas K., Goumas G., and Koziris N., *Performance analysis of concurrent red-black trees on htm platforms*, TRANSACT, 2015.

[6] Siakavaras D., Nikas K., Goumas G., and Koziris N., *Massively Concurrent Red-Black Trees with Hardware Transactional Memory*, 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), 2016.

[7] R. Bayer and M. Schkolnick., *Readings in database systems*, ch. Concurrency of Operations on B-trees, pp. 129–139, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1988.

[8] Tudor, D., Guerraoui, R., Trigonakis, V., *Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures*, In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, Pages 631-644. ASPLOS 2015.

[9] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun., *A Practical Concurrent Binary Search Tree*, PPoPP 2010.

[10] Drachsler D., Vechev, M.T., Yahav, E., *Practical concurrent binary search trees via logical ordering*, In: PPoPP, pp. 343-356. ACM(2014).

[11] Aravind Natarajan and Neeraj Mittal., *Fast Concurrent Lock-free*, Binary Search Trees. PPoPP 2014.

[12] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel., *Non-blocking Binary Search Trees*, PODC 2010.

[13] Damron P., Fedorova A., Lev Y., Luchangco V., Moiv M., Nussbaum D., *Hybrid transactional memory*, In: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems, Pages 336-346. ASPLOS 2006.

[14] M. Moir., *Hybrid transactional memory*, July 2005.

[15] Casper J., Oguntebi T., Hong S., Bronson N., Kozyrakis C., Olukotun k., *Hardware acceleration of transactional memory on commodity systems*, In: Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems, Pages 27-38. ASPLOS 2011.

[16] A. Shriraman, M. F. Spear, H. Hossain, V. J. Marathe, S. Dwarkadas, and M. L. Scott., *An integrated hardware-software approach to flexible transactional memory*, SIGARCH Computer Architecture News, 35, June 2007.

[17] J. R. Larus and R. Rajwar., *Transactional Memory. Synthesis Lectures on Computer Architecture.*, Morgan & Claypool, 2007.

[18] Dijkstra, E. W., *A note on two problems in connection with graphs.*, Numerische Mathematik 1: 269–271. doi:10.1007/BF01386390, 1959.

[19] R. C. Prim, *Shortest connection networks and some generalizations.*, In: Bell System Technical Journal, 36 (1957), pp. 1389–1401.

[20] N. Anastopoulos, K. Nikas, G. Goumas, and N. Koziris, *Early experiences on accelerating dijkstra's algorithm using transactional memory.*, in Proc. 3rd Workshop on Multithreaded Architectures and Applications (MTAAP'09), 2009.

[21] K. Nikas, N. Anastopoulos, G. Goumas, N. Koziris, *Employing transactional memory and helper threads to speedup Dijkstra's algorithm.*, Parallel Processing, 2009. ICPP'09. International Conference on, 388-395.

[22] Crain, T., Gramoli, V., Raynal, M., *A speculation-friendly binary search tree*, In: PPoPP '12, Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming Pages 161-170, ACM New York, 2012.

[23] Chatterjee, B., Nguyen, N., Tsigas, P., *Efficient lock-free binary search trees*, In: PODC '14, Proceedings of the 2014 ACM symposium on Principles of distributed computing Pages 322-331, ACM New York, 2014.

[24] Prokopec, A., Bronson, N., Bagwell, P., Odersky, M., *Concurrent tries with efficient non-blocking snapshots*, In: PPoPP '12, Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming Pages 151-160 , ACM New York, 2012.

[25] Crain, T., Gramoli, V., Raynal, M., *A contention-friendly binary search tree*, In: Euro-Par'13, Proceedings of the 19th international conference on Parallel Processing Pages 229-240, Springer-Verlag Berlin, Heidelberg, 2013.

[26] Natarajan, A., Savoie, L., Mittal, N., *Concurrent Wait-Free Red Black Trees*, In: SSS 2013 Proceeding of the 15th International Symposium on Stabilization, Safety, and Security of Distributed Systems - Volume 8255 Pages 45-60, Springer-Verlag New York, 2013

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Μελέτη και Αξιολόγηση Τεχνικών Παραλληλοποίησης Δομών Δεδομένων και Αλγορίθμων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Χριστίνα Χρ. Γιαννούλα

Επιβλέπων: Γεώργιος Γκούμας
Λέκτορας Ε.Μ.Π.

Αθήνα, Ιούλιος 2016

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑ-ΤΩΝ

Μελέτη και Αξιολόγηση Τεχνικών Παραλληλοποίησης Δομών Δεδομένων και Αλγορίθμων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Χριστίνα Χρ. Γιαννούλα

Επιβλέπων: Γεώργιος Γκούμας
Λέκτορας Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 14η Ιουλίου 2016.

....	....	....
Γ. Γκούμας Λέκτορας Ε.Μ.Π.	Ν. Κοζύρης Καθηγητής Ε.Μ.Π.	Κ. Σαγώνας Αναπληρωτής Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2016.

....

Χριστίνα Χρ. Γιαννούλα
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © Χριστίνα Γιαννούλα, 2016. Εθνικό Μετσόβιο Πολυτεχνείο.
Με επιφύλαξη κάθε δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ' ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπο την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τη συγγραφέα.
Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τη συγγραφέα και δεν πρέπει να ερμηνευτεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Στις μέρες μας, οι πολυπύρηνοι επεξεργαστές έχουν γίνει η κυρίαρχη πλατφόρμα υπολογισμών και έχουν εισαχθεί σε πολλά προγραμματιστικά περιβάλλοντα. Ο παράλληλος προγραμματισμος δεν αφορά πλέον μόνο επιστημονικές εφαρμογές που τρέχουν σε υπερυπολογιστές, αλλα καλύπτει επίσης ένα μεγάλο φάσμα εφαρμογών για προσωπικούς υπολογιστές. Ένα από τα πιο δύσκολα προβλήματα στα συστήματα παράλληλης επεξεργασίας είναι η ανάπτυξη παράλληλου λογισμικού το οποίο κλιμακώνει αποδοτικά. Αρκετές εφαρμογές δεν κλιμακώνουν μετά από έναν αριθμό επεξεργαστών εξαιτίας του αυξημένου κόστους επικοινωνίας. Προκειμένου να αξιοποιηθούν οι διαθέσιμες αρχιτεκτονικές, οι βασικές δομές δεδομένων και οι σειριακοί αλγόριθμοι πρέπει να επανασχεδιασθούν. Το πρώτο μέρος αυτής της διπλωματικής αφορά τις παράλληλες δομές δεδομένων, με ιδιαίτερη έμφαση στα δυαδικά δέντρα αναζήτησης, εξετάζοντας τον τρόπο συγχρονισμού τους, τα ιδιαίτερα χαρακτηριστικά τους και την κλιμακωσιμότητα που προσφέρουν. Στο δεύτερο μέρος της διπλωματικής παρουσιάζεται μια παραλληλοποίηση του αλγορίθμου του Dijkstra που είναι ένας σειριακός αλγόριθμος. Αυτή η υλοποίηση χρησιμοποιεί Transactional Memory, για να συντονίσει αποτελεσματικά τις ταυτόχρονες προσβάσεις των νημάτων στις κοινές δομές δεδομένων και την έννοια των Helper Threads, για να εξάγει παραλληλισμό. Η αξιολόγηση του αλγορίθμου γίνεται σε ένα σύστημα που υποστηρίζει Hardware Transactional Memory.

Λέξεις-Κλειδία: παράλληλες δομές δεδομένων, δυαδικά δέντρα αναζήτησης, κλιμακωσιμότητα, παράλληλος προγραμματισμός, αλγόριθμος του Dijkstra, Helper Threads, Hardware Transactional Memory

Abstract

Nowadays, multicore processors have become the dominant computing platform and are being used by many programming environments. Parallel programming is no longer about scientific applications run in supercomputers, but covers a wider range of applications on personal computers, too. The most difficult problem is to develop parallel software that scales efficiently. Several applications do not scale further than a number of processors due to communication overhead. To exploit the available architectures basic data structures and sequential algorithms must be redesigned. In the first part of this thesis we study concurrent data structures, particularly focusing on concurrent binary search trees, with respect to the way they are synchronized, their special characteristics and the scalability they provide. The second part of this thesis presents a parallelization of the inherently serial Dijkstra's algorithm. This implementation employs Transactional Memory to efficiently orchestrate the concurrent thread's accesses to shared data structures and the concept of Helper Threads to extract parallelism. We evaluate the execution of the algorithm on a system that supports Hardware Transactional Memory.

Keywords: Concurrent Data Structures, Binary Search Trees, scalability, parallel programming, Dijkstra's algorithm, Helper Threads, Hardware Transactional Memory

Ευχαριστίες

Η παρούσα διπλωματική εργασία εκπονήθηκε στο Εργαστήριο Υπολογιστικών Συστημάτων της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου, υπό την επίβλεψη του Λέκτορα Ε.Μ.Π. Γεώργιου Γκούμα.

Θα ήθελα να ευχαριστήσω τον καθηγητή μου κ. Γεώργιο Γκούμα, για την εποπτεία κατά την εκπόνηση της εργασίας μου, τις γνώσεις που μου προσέφερε με τη διδασκαλία του και την ευκαιρία που μου έδωσε να δουλέψω στο εργαστήριο.

Θα ήθελα επίσης να ευχαριστήσω τον καθηγητή της σχολής κ. Νεκτάριο Κοζύρη για την έμπνευση που μου προσέφερε με τη διδασκαλία του και να τον συγχαρώ για υψηλό επίπεδο σπουδών των μαθημάτων διδασκαλίας του και του εργαστηρίου.

Ιδιαίτερα θα ήθελα να ευχαριστήσω τον Υποψήφιο Διδάκτωρ Δημήτριο Σιακαβάρα για τη συνεχή καθοδήγησή του κατά τη διάρκεια εκπόνησης αυτής της διπλωματικής εργασίας, καθώς και για την ενθάρρυνσή του, την υπομονή του και το χρόνο που αφιέρωσε. Χωρίς τη συμβολή του η ολοκλήρωση αυτής της εργασίας δε θα ήταν εφικτή.

Επιπλέον, επιβάλλεται να ευχαριστήσω τους φίλους μου και τους συμφοιτητές μου για την πολύτιμη βοήθειά τους σε επιστημονικό και προσωπικό επίπεδο.

Τέλος, θα ήθελα να ευχαριστήσω θερμά την οικογένειά μου, τους γονείς μου και την αδερφή μου για την αγάπη, την αμέριστη υποστήριξή τους όλα αυτά τα χρόνια και την εμπιστοσύνη τους σε κάθε μου επιλογή.

Χριστίνα Γιαννούλα

Contents

1 Introduction **19**
- 1.1 Overview . . . 19
- 1.2 Parallel Architecture and Parallel Programming . . . 20
  - 1.2.1 Memory coherence . . . 20
  - 1.2.2 Memory consistency . . . 21
  - 1.2.3 Amdahl's law . . . 22
- 1.3 Parallel Architectures . . . 23
  - 1.3.1 Shared memory architecture . . . 25
  - 1.3.2 Distributed memory architecture . . . 26
  - 1.3.3 Hybrid memory architecture . . . 27
- 1.4 Synchronization . . . 27
  - 1.4.1 Synchronization definition . . . 27
  - 1.4.2 Synchronization techniques . . . 29

2 Concurrent Search Trees **33**
- 2.1 Concurrent Data Structures . . . 33
- 2.2 Search Trees . . . 34
  - 2.2.1 Binary Search Trees . . . 35
  - 2.2.2 AVL Trees . . . 36
  - 2.2.3 Red-Black Trees . . . 37
- 2.3 Techniques for constructing concurrent data structures . . . 43
  - 2.3.1 Coarse-grained locking . . . 43
  - 2.3.2 Fine-grained locking . . . 45
  - 2.3.3 Lock-free programming . . . 45
- 2.4 Basic Interface in Concurrent Search Trees . . . 46
- 2.5 A naive approach . . . 47
  - 2.5.1 Description . . . 48
  - 2.5.2 Implementation details . . . 49
- 2.6 A more sophisticated approach . . . 52
  - 2.6.1 Description . . . 52
  - 2.6.2 Implementation details . . . 58
- 2.7 Evaluation . . . 58
  - 2.7.1 System Configuration . . . 58

2.7.2 Run Configurations 58
2.7.3 Results 60

3 Transactional Memory **65**
3.1 Transactional Memory (TM) 65
3.1.1 Software Transactional Memory (STM) 66
3.1.2 Hardware Transactional Memory (HTM) 66
3.1.3 Hybrid Transactional Memory 67
3.2 Basic TM Characteristics 68
3.2.1 Real TM implementations 73
3.3 Intel's Haswell HTM 73
3.3.1 Transactional Synchronizations Extensions (TSX) 74

4 A parallelization of Dijkstra's algorithm **81**
4.1 Dijkstra's algorithm 81
4.1.1 Algorithm's history 81
4.1.2 Description 82
4.1.3 Complexity 83
4.1.4 Proof of correctness 84
4.1.5 Applications 85
4.2 The concept of Helper Threads 85
4.3 Parallelizing Dijkstra's algorithm 86
4.3.1 Introduction 86
4.3.2 The algorithm 87
4.3.3 Optimizations 90
4.4 System Configuration 93
4.5 Experimentation 94
4.5.1 Experimentation in the serial algorithm 94
4.5.2 Experimentation in the parallel algorithm 97
4.6 Results 103
4.6.1 Performance results 103
4.6.2 A closer look at the results 104
4.6.3 Experimentation in padding technique 108
4.7 Employing skip list 111
4.7.1 The skip list structure 111
4.7.2 Comparison with the binary heap 112
4.7.3 Results 116

5 Conclusion and Future Work **121**

List of Figures

1.1 MESI state diagram. The transitions are labeled "action observed / action performed". 21

1.2 Total speedup of a parallel program as parallel fraction and number of processors change. 23

1.3 Flynn's taxonomy. 24

1.4 Classic organization of a SMP. 25

1.5 Classic organization of a Distributed Memory Architecture. 26

1.6 Classic organization of a Hybrid Memory Architecture. 27

2.1 Search data structure interface. Updates have two phases: a parse phase, followed by a modification phase. 34

2.2 An example of a tree in external and internal format. 35

2.3 A right and a left tree rotation. 37

2.4 Left left case. T1, T2, T3 and T4 represent subtrees which are themselves balanced AVL trees. 37

2.5 Right right case. T1, T2, T3 and T4 represent subtrees which are themselves balanced AVL trees. 38

2.6 Left right case. T1, T2, T3 and T4 represent subtrees which are themselves balanced AVL trees. 38

2.7 Right left case. T1, T2, T3 and T4 represent subtrees which are themselves balanced AVL trees. 38

2.8 An example of a red–black tree. 39

2.9 Case 1 upon an insetion of a node z in a Red-Black tree. The code for case 1 changes the colors of some node to preserve the coloring properties. . . . 41

2.10 Case 2 and case 3 upon an insetion of a node z in a Red-Black tree. We transform case 3 into case 2 by a left rotation. Case 2 causes some color changes and a right rotation to preserve the coloring properties. 42

2.11 The cases upon a deletion of a node in a Red-Black tree. Darkened nodes have color attributes black, heavily shaded nodes have color attributes red, and lightly shaded nodes have color attributes represented by c and c', which may be either red or black . The letters α, β, ..., ζ represent arbitrary subtrees. Each case transforms the configuration on the left into the configuration on the right by changing some colors and/or performing a rotation. Any node pointed to by x has an extra black and is either doubly black (when a black node is deleted and replaced by a black child, the child is marked as doubly black) or red-and-black. (a) Case 1 is transformed to case 2, 3, or 4 by exchanging the colors of nodes B and D and performing a left rotation. (b) In case 2, the extra black represented by the pointer x moves up the tree by coloring node D red and setting x to point to node B. (c) Case 3 is transformed to case 4 by exchanging the colors of nodes C and D and performing a right rotation. (d) Case 4 removes the extra black represented by x by changing some colors and performing a left rotation (without violating the coloring properties). This figure has been taken from [5]. 44
2.12 A typical bst node structure in memory using padding, such that to be the same bytes as one cache line. 50
2.13 Deletion of node D in an internal tree. While searching D's successor to swap their key-value pairs, the subtree rooted at D node must be locked. . 51
2.14 The platform used in evaluation of concurrent search trees. 59
2.15 Throughput of concurrent naive implementations for 2K key range and the three workloads. 60
2.16 Throughput of concurrent naive implementations for 32K key range and the three workloads. 61
2.17 Throughput of concurrent naive implementations for 2000000 key range and the three workloads. 61
2.18 Throughput of concurrent sophisticated implementations for 2K key range and the three workloads. 63
2.19 Throughput of concurrent sophisticated implementations for 32K key range and the three workloads. 63
2.20 Throughput of concurrent sophisticated implementations for 2000000 key range and the three workloads. 63

3.1 A simple example of eager versioning. 69
3.2 A simple example of lazy versioning. 69
3.3 A discussion of pessimistic detection. 71
3.4 A discussion of optimistic detection. 71
3.5 Transaction's status is captured to EAX register's bits. 78
3.6 A parallel execution of two threads using RTM that results to coherence problems. 79

4.1 Early history of shortest paths algorithms. 82

4.2 Proof of corectness of Dijkstra's algorithm. When a vertex u is added to S set (visited nodes), then $dist[u] = [s, u]$. 84
4.3 Applications of Dijkstra's algorithm. 86
4.4 Execution pattern of the HT scheme. 88
4.5 The platform used in evaluation of Dijkstra's algorithm. 94
4.6 Evaluation of the four phases of the algorithm in different graphs. 96
4.7 Experimentation in padding technique on the serial algorithm. 97
4.8 Time elapsed in random node-1M-edge-10M graph for different number of possible retries per transaction of the main thread. 98
4.9 Time elapsed in rmat node-10M-edge-500M graph for different number of possible retries per transaction of the main thread. 98
4.10 Time elapsed in random node-1M-edge-10M graph for different number of possible retries per transaction of helper threads. 100
4.11 Time elapsed in rmat node-10M-edge-500M graph for different number of possible retries per transaction of helper threads. 100
4.12 Time elapsed in random node-1M-edge-100M graph for different number of neighbors examined for relaxation per transaction of the main thread. . 101
4.13 Time elapsed in rmat node-10M-edge-500M graph for different number of neighbors examined for relaxation per transaction of the main thread. . 101
4.14 Time elapsed in random node-1M-edge-100M graph for different number of neighbors examined for relaxation per transaction of helper threads. . . 102
4.15 Time elapsed in rmat node-10M-edge-500M graph for different number of neighbors examined for relaxation per transaction of helper threads. . . 102
4.16 Multithreaded speedups for graphs of different density. 104
4.17 Distribution of relaxations between the main and helper threads. 105
4.18 The number of main thread's commits/aborts. 106
4.19 The percentage of total transactional aborts for all threads in total number of transactions. 107
4.20 Distribution of time spent in different phases of main thread's execution. . 108
4.21 Despite not using padding, simultaneous relaxations are performed in nodes that reside in different cache lines. 109
4.22 Experimentation in padding technique on the parallel algorithm. 110
4.23 An example of a skip list. 111
4.24 Time elapsed in serial execution for the three different structures used for the priority queue. 113
4.25 Time elapsed in parallel execution for the three different structures used for the priority queue. 114
4.26 Distribution of relaxations between the main and helper threads for the random-node-10M-edge-500M graph using two different structures for the priority queue. 114
4.27 The number of commits/aborts of the main thread in parallel execution for the three different structures used for the priority queue. 116

4.28 The number of commits/aborts of helper threads in parallel execution for the three different structures used for the priority queue. 116
4.29 Multithreaded speedups for graphs of different density when using our optimized skip list. 117
4.30 An example of binary heap. The vertices with the red color have not obtained their optimal distance from the source. 118
4.31 A speedup approximation based on main thread's relaxations in case of 14 threads and the speedup achieved using the binary heap structure. 118
4.32 A speedup approximation based on main thread's relaxations in case of 14 threads and the speedup achieved using the optimized skip list structure. . 119

Listings

1.1 Inconsistencies due to lack of synchronization 28
2.1 A typical bst node structure 50
3.1 Example: elision of a TAS lock 76
3.2 Example: adding global lock to transaction's read set 78
3.3 An RTM example 78
4.1 Dijkstra's algorithm 83
4.2 Main thread's code. 89
4.3 Helper threads' code. 90
4.4 Main thread's code for a real HTM. 92
4.5 Helper threads' code for a real HTM. 92
4.6 The four phases of the algorithm. 95

Chapter 1

Introduction

1.1 Overview

In 1965, Moore predicted that the number of transistors in an integrated circuit will double approximately every two years, resulting to exponential increase in raw computer power. However, the increasing number of transistors per processor set physical limitations. The dense chips use more electric power and generate more heat limiting the evolution in processors.

As Moore's law was used in the semiconductor industry, more and more transistors integrated in the same silicon chip and the chip performance doubled every 18 months. These high performance microprocessors named as multicore processors. Nowadays, multicore processors is a characteristic of supercomputers, distributed systems and personal computers. Increasing the number of processors can result to high-performance computing. However, multiprocessors systems cause communication and synchronization problems and the scalability of multiprocessors systems remains a challenge as the number of processors increase.

At the same time, in order to take full advantage of these available hardware resources, computer industry developed new architectures techniques. For example, the use of deeper pipelines superscalar architectures increased the operation throughput. As a result, conventional architectures replaced by parallel architectures with a view to maximize performance of applications.

The multicore systems constitute a solution to computation-intensive applications. Scientific applications such as astrophysical and cosmological simulations, weather forecasting, applications in industry and other sectors, user applications such as search engines and web servers are computationally demanding. Contemporary serial algorithms, even if optimized, cannot achieve an optimum performance when executed on multiprocessors architectures. As a consequence, sequential algorithms must be redesigned such that run in parallel and exploit the available architectures.

Parallel programming demands synchronization among parallel processes or threads in order to avoid conflicts and race conditions. Synchronization techniques are implemented

in both software and hardware. However, communication and synchronization between different subtasks are some of the greatest obstacles to getting good parallel performance. The challenge is to create hardware and software that will make it easy to develop parallel processing programs such that to achieve good performance and scalability as the number of cores per chip increase.

1.2 Parallel Architecture and Parallel Programming

1.2.1 Memory coherence

Modern processing systems consist of multiple level of cache memory, which reduce the cost of multiple references in the same memory location as there exist copies of recently uses memory locations close to the processor such that to have a quick and low cost access to them. Though, the existence of copies of the same memory location in different levels of cache memory cause many problems. For example, in a single processor system when a device uses Direct Memory Access (DMA) to read data from main memory, any changes to that memory residing in processor caches must be flushed out first.

In multiprocessor systems, it is possible to have many copies of the same memory location in multiple caches when several processors access to this memory location simultaneously. Provided none of the processors changes the data in this location (read only), they can share it without any problem. But as soon as one updates the location, all the other must be notified of the update, otherwise they might work on an out-of-date copy, that resides in their local cache. As a consequence, it must be followed a scheme that ensures that none of the processors is accessing a stale value of a memory location. This scheme is known as memory coherence protocol. A practical multiprocessor invalidate protocol, the most widely used, which attempts to minimize bus usage is the MESI protocol.

In MESI protocol, any cache line can be marked with one of the following states:

Modified: The cache line is present only in the current cache (the only cache copy), and is dirty, it has been modified from the value in main memory. The cache is required to write data back to main memory at some time in the future, before permitting any other read of the (no longer valid) main memory state.

Exclusive: The cache line is present only in the current cache (the only cache copy) and matches the main memory. In a read request from another processor, the cache line will change to the Shared state and in a write request from that processor will change to the Modified state.

Shared: The cache line may be stored in other caches of the machine and matches the main memory.

Invalid: The cache line is not valid.

The MESI protocol uses the above mentioned states for every cache line to permit all processor to have a consistent view on all memory locations. The cache line changes state, according to the state diagram depicted in figure 1.1, as a function of memory access events. An event may be due to local processor activity or due to bus activity.

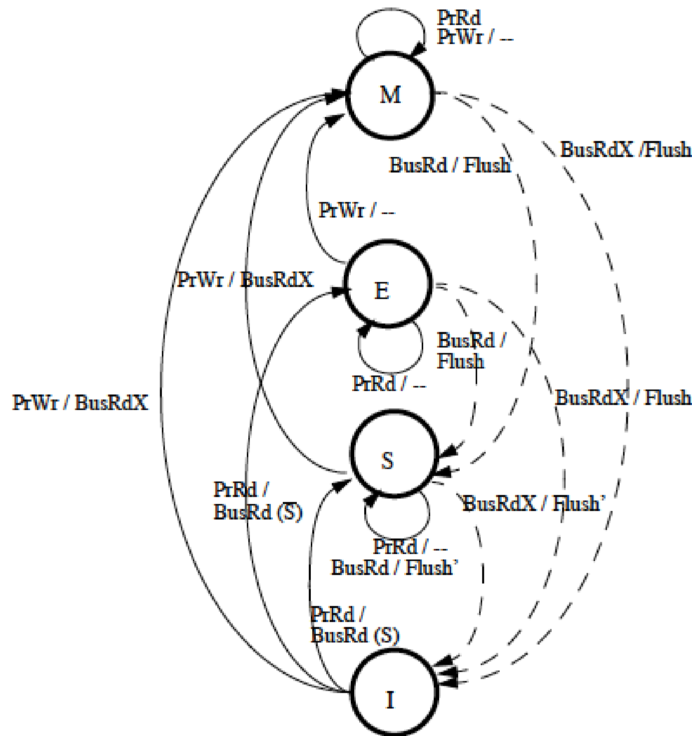


Figure 1.1: MESI state diagram. The transitions are labeled "action observed / action performed".

1.2.2 Memory consistency

As mentioned above, cache coherence means that all the processors see the same value for a particular memory address as they should have if there were no caches in the system. Memory consistency on the other hand, ensures that all the memory instructions appear to execute in the program order, that is consistency refers to the order of accesses to all memory locations. It defines how all the memory instructions in a multiprocessor system will be ordered.

In multiprocessors systems the order between operations of a program is not always guaranteed. Operations issue concurrently by many processors in an order that cannot be determined in advanced. Most real hardware use first-in-first-out (FIFO) write buffers and in this way there is not guaranteed order in parallel programs.

A memory consistent model is a specification of the allowed behaviour of parallel programs executing with shared memory. In a parallel program, unlike to a single-threaded execution, multiple correct behaviours are usually allowed. There are different memory consistencies such as sequential consistency, relaxed consistency and might have an impact on the final program result.

1.2.3 Amdahl's law

Sequential algorithms need to be redesigned in order to run in parallel and exploit the available hardware and processors. In parallel algorithms, large problems are divided into smaller ones, which are then solved in the same time, as the available cores execute concurrently smaller problems. In comparison with sequential algorithms, it is harder to develop parallel algorithms, however the parallel program have better performance and scalability.

Amdahl's law [1] is a theoretical formula that gives the maximum theoretical speedup achieved. Speedup is a measure of how many times the parallel program is faster than the best sequential algorithm. If T_s is the runtime of the fastest sequential program and T_p is the runtime of the parallel version of the program on p processors, the speedup is defined as:

$$Speedup(S) = \frac{T_s}{T_p}$$

Typically, the speedup S is related with the number of processors p in the inequality: $S \leq p$. If $S = p$, the speedup is *linear*.

Amdahl's law show that the theoretical speedup increases by improving a part of the total program. Consider f the fraction of the problem, which cannot be parallelized and must be executed sequentially, then the runtime of the parallel program is:

$$T_p = fT_s + \frac{(1-f)T_s}{p}$$

and the expression for the speedup is:

$$\text{Total Speedup } S = \frac{1}{f + \frac{1-f}{p}}$$

Amdahl's law is often used to predict the theoretical speedup when using multiple processors. As the number of processors p goes to infinity, the total speedup goes to $1/f$. The theoretical speedup is limited by the part of the program that cannot be executed in parallel. If the parallel part of a program is relatively small, its speedup would be equally small. For instance, the fraction is $f = 90\%$, the parallel program can be 10 times faster in the best case than the serial program, independently of the number of processors. Figure 1.2 depicts the total speedup in parallel executions of different sequential fractions f and number of processors p. According to this figure, the total speedup of a program is bounded by the sequential part of the program and using more processors does not increase the speedup in all cases. Parallelizing more and more of the sequential program is the solution to maximize the performance.

Scalability, another measure for performance, can refer to the capability of a system or a program to increase its performance when more processors are added. Assumed that the program have a constant size, the runtime of the program is expected to scale up as the number of processors increases. However, there are many factors that limit the scalability of a program. First of all, the program has to be divided into small equal pieces, each of

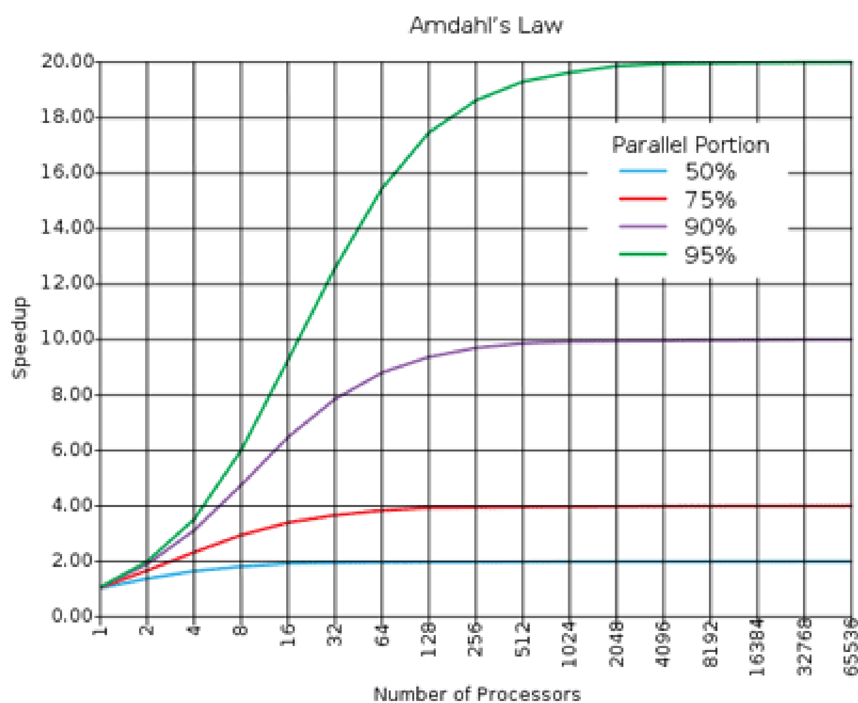


Figure 1.2: Total speedup of a parallel program as parallel fraction and number of processors change.

them is executed in a separate processor. If the pieces are not equal, the processors would wait for other ones with the larger pieces to terminate. Furthermore, the scalability can be limited due to the time spent in communication and synchronization among processors. If this communication and synchronization time is significant compared with the total time, the program does not scale up when the number of processors increases. To conclude, load balancing and time spent for synchronization and communication restrict significantly the scalability of a program and must be taken into consideration in parallel programming.

1.3 Parallel Architectures

Flynn's taxonomy [2] distinguishes computer architectures, according to the level of parallelism they employ to process instructions and data streams. There are four categories:

- **SISD:** Single Instruction, Single Data
  A sequential computer. Sequential computers are incapable of performing parallel operations.

- **SIMD:** Single Instruction, Multiple Data

A parallel computer with a single instruction stream, which performs the same instruction on multiple data.

- **MISD:** Multiple Instruction, Single Data
  Multiple processing units perform tasks-instructions on the same data. MISD is an uncommon and non-commercial architecture.
- **MIMD:** Multiple Instruction, Multiple Data
  A parallel computer, in which each processor executes independent instruction streams on independent data streams. This architecture is the most common and widely used form of parallel architecture. Two examples of this architecture are clusters and systems of multicore processors.

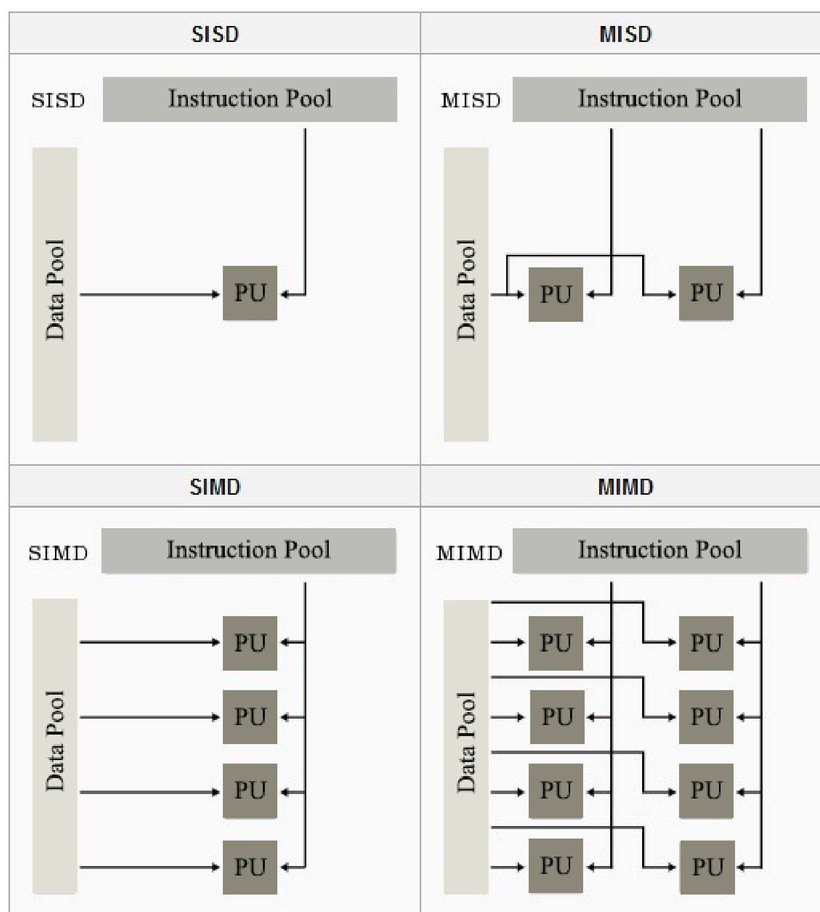


Figure 1.3: Flynn's taxonomy.

As mentioned above, MIMD multiprocessing architecture is the most common parallel architecture and suitable for a wide variety of tasks. MIMD architectures can be catego-

rized based on their memory organization in three categories that will be further analyzed in the next sections: shared memory architectures, distributed memory architectures and hybrid architectures.

1.3.1 Shared memory architecture

In shared memory architecture, each processor has each own private cache memory hierarchy and all processors share a single physical space, known as global memory. A single system bus interconnects all processors. Such systems can execute independent tasks, which have their own virtual address spaces, even if they share a physical address space. Processors communicate by sharing variables stored in the global memory and can access any memory location via loads and stores. If the access to any memory location takes the same amount of time to all processors, the memory organization is called symmetric multiprocessor (SMP) and can be viewed in figure 1.4.

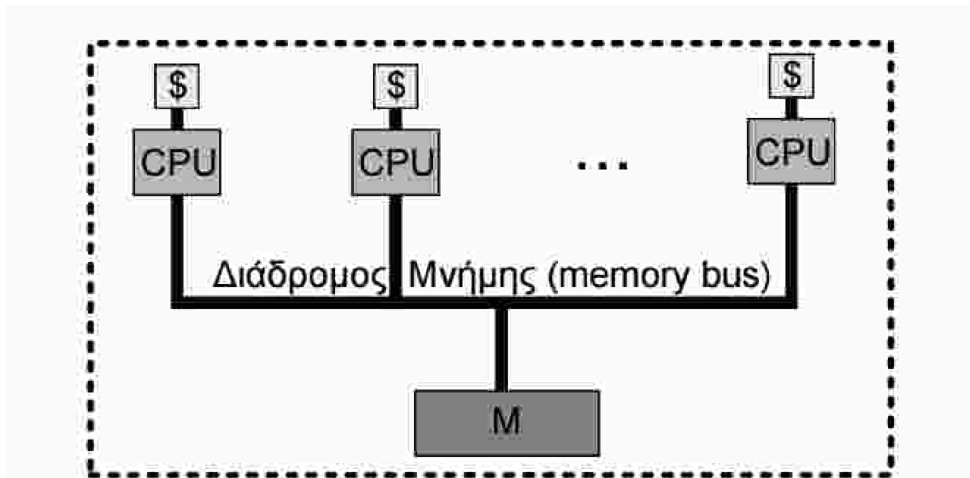


Figure 1.4: Classic organization of a SMP.

Shared memory architecture come in two memory organizations. If the amount of time taken to access any global memory address is equal independently which processor requests the access, the memory organization is called Uniform Memory Access (UMA). If some memory accesses are much faster than the others, depending on which processor requests for which global memory address, the memory organization is called Non-Uniform Memory Access (NUMA). NUMA architectures can have lower latency to nearby memory and higher memory bandwidth.

Processors in shared memory architectures can operate tasks in parallel using the same shared data and race conditions may occur. As a result, processors need to coordinate in order to avoid concurrent accesses to shared data. Thus, synchronization mechanisms like locks and atomic variables, are used in such situations. Furthermore, as analyzed in

the previous section, cache coherent protocols like MESI, impose a universal sequence of accesses to the global memory.

The efficient access to all shared data from any processor via simple load and store operations on them make shared memory architectures attractive for parallel programming. However, this memory organization has a single system bus that interconnects all processors and a limited memory bandwidth. As a result, it can be used for no more than 20 or 30 processors because of the limited single bus and the limited memory bandwidth.

1.3.2 Distributed memory architecture

In distributed memory architectures each processor has a local cache hierarchy and a local main memory. The processors are connected in an interconnection network (e.g Ethernet) and are called nodes. They have not shared memory addresses and the only way to communicate each other is via message passing through the interconnection network. The system provides to the programmer routines for send/receive messages to/from any processor. The figure 1.5 depicts the organization of a distributed memory architecture.

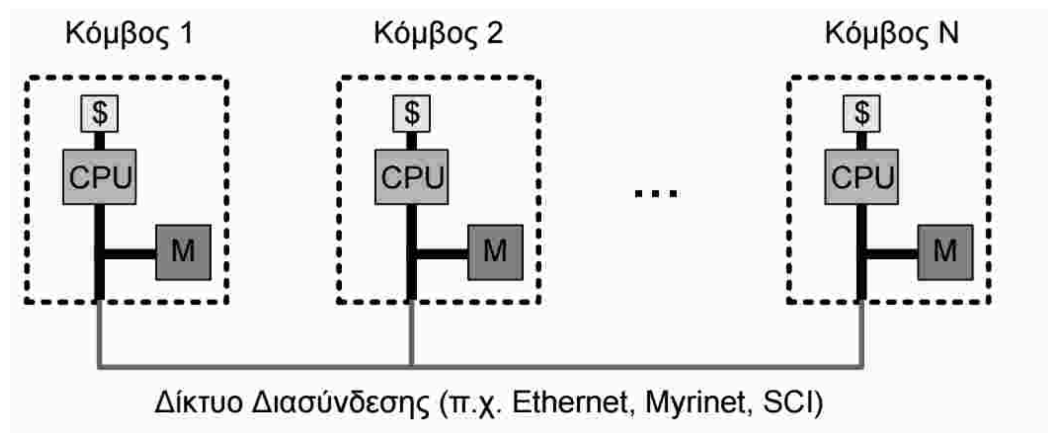


Figure 1.5: Classic organization of a Distributed Memory Architecture.

This architecture is used in clusters that are generally collections of commodity computers that are connected to each other over an I/O interconnect in a network. Each processor has a separate copy of the operating system.

One drawback of clusters is the management cost. The management cost of a cluster with n nodes equals to the management cost of n computers, while the management cost of a multiprocessor with n cores is the same management cost as one single computer. Moreover, another drawback of clusters is the bandwidth and the latency. The processors in a cluster are connected using the I/O interconnection, while the cores in a multiprocessor system are connected via memory interconnect. The memory interconnect has higher bandwidth and lower latency and allow better communication performance.

Programming in a distributed memory architecture is a challenge, since every communication must be identified in advance. Efficient parallelization requires understanding the memory dependencies of the program and an effective distribution of memory in advance in order to eliminate communication between remote processors.

Finally, distributed systems can achieve high scalability because of the absence of shared memory and race conditions. They are constructed of thousand independent nodes that can be dynamically inserted and removed from the network.

1.3.3 Hybrid memory architecture

The hybrid architecture combines the previous two architectures and takes advantages of their benefits. A hybrid system is like a distributed system, in which a symmetric multiprocessor has taken the place of each single processor node. Figure 1.6 depicts the organization of a hybrid architecture. This typical architecture is used in clusters and supercomputers, favors parallel processing within each node and scales up as a distributed memory architecture.

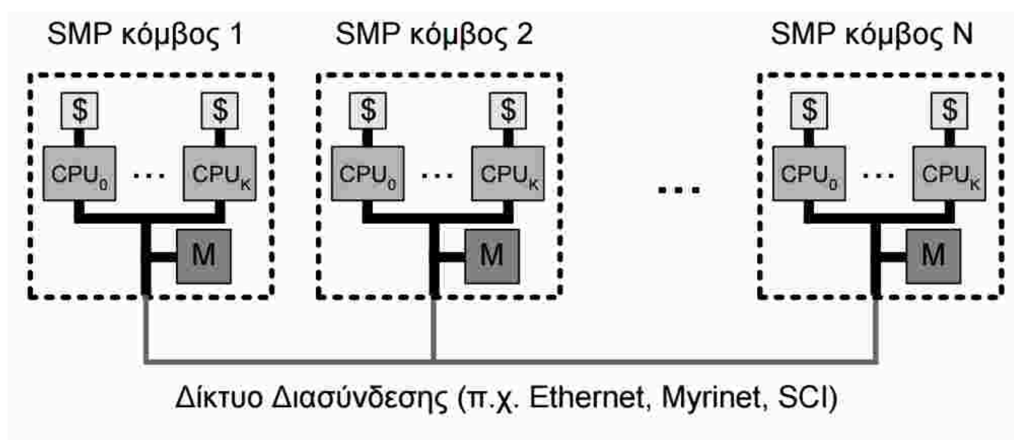


Figure 1.6: Classic organization of a Hybrid Memory Architecture.

1.4 Synchronization

1.4.1 Synchronization definition

In parallel programming, threads or processes * need to communicate with each other and execute operations in the same shared data or variables. The operations or the code

*We will use the term process to refer to a parallel task, but in parallel programming we can have either processes or threads.

of a process must be executed as if the processes were running in isolation, each with access to its own memory space. However, when processes run in parallel and perform operations on common data structures, they have to be synchronized, otherwise the result will be undefined. The synchronization is extremely important in parallel programming.

Process synchronization is defined as a technique that more concurrent processes do not simultaneously execute some particular segment of the program known as critical section. The critical section is a serialized segment of the program. When one process starts executing the critical section, the other processes should wait until the first completes the critical section. If synchronization techniques are not applied, the values of the variables may be unpredictable and vary depending on the timings of context switches.

A classic problem of synchronization is the Readers-Writers problem, which deals with situations in which many processes try to access the same shared resource at the same time. Some processes may read and some may write, with the constraint that no process may access the share data for either reading or writing, while another process is writing on it. It is only allowed for two or more readers to access the share data concurrently. Another example of how the absence of synchronization among processes or threads leads to inconsistencies can be viewed in listing 1.1. The counter is incremented once by one thread and is decremented once by the other. When the threads execute the critical segment concurrently, the final result of the counter can have either the same initial value or a decreased or increased value.

Listing 1.1: Inconsistencies due to lack of synchronization

```
1  Thread 1                   Thread 2
2  var1 = counter;            var2 = counter;
3  var1 += 1;                 var2 -= 1;
4  counter = var1;            counter = var2;
```

Other than mutual exclusion synchronization can also deals with the following:

- deadlock, is a situation in which many processes are waiting for a shared resource which is being held by another process and there is no progress in the program.
- starvation, which occurs when a process is perpetually denied necessary resources to enter the critical section and the process is forced to wait indefinitely.
- priority inversion, which occurs when a high priority task is in the critical section and it may be interrupted by a medium priority task.
- busy waiting, is a situation in which a process repeatedly checks to determine if it has access to a critical section. This spinning can generate an arbitrary time delay.

Accesses to critical section by different processes are controlled by using synchronization schemes. They can be divided in two categories, blocking synchronization and non-blocking synchronization.

Blocking synchronization

- deadlock-free
  Using mutual exclusion[*], it guarantees that some processes will finish their task in a finite number of steps.
- starvation-free
  Using mutual exclusion, it guarantees that every process will finish their task in a finite number of steps.

Non-blocking synchronization

- lock-free
  It guarantees that some processes will finish their task in a finite number of steps.
- wait-free
  It guarantees that every process will finish their task in a finite number of steps.

Non-blocking schemes allow access by multiple concurrent processes without mutual exclusion. Multiple processes access shared resources and perform operations on them without blocking. The consistency in shared resources is guaranteed using individual operations.

1.4.2 Synchronization techniques

There are different synchronization techniques:

Mutual exclusion

Mutual exclusion is the most usual technique to achieve blocking synchronization. It ensures that two or more concurrent processes are not in their critical section at the same time. Only one will access the critical section of the program at a given time. While one process executes operations in the shared resources, all other should be kept waiting and when that process has finished its work in the shared data, one of the processes waiting will proceed. Mutual exclusion is implemented via mechanisms like semaphores, mutexes and locks. Almost all locks use Test-And-Set (TAS) atomic operation to set a memory location. The process that sets the lock to the LOCKED state is considered to be the lock owner and can proceed to the critical section.

The operation TAS is atomic and only one process can set the memory location at a time. If the shared memory location is in the UNLOCKED state the process can set it as LOCKED, become the lock owner and proceed. Otherwise, if the shared memory location is in the LOCKED state the process continues to loop while checking the state until it becomes UNLOCKED and successfully acquires the lock. This continuously executing TAS on the shared memory location causes heavy bus traffic. In TAS a process sets the memory location of the lock and checks if the previous state of the lock was LOCKED or UNLOCKED in order to proceed to the critical section or wait. According to cache coherence protocols, when a TAS operation sets the memory location (lock), all other copies of lock will be invalidated, including the owner's. This causes heavy cache coherence

[*]It is further analyzed in the next subsection.

protocol traffic. As a result, an improvement of TAS is Test-and-Test-And-Set (TTAS) operation, in which the state of the lock is first read locally and the process tries to set the lock with TAS only if it appears to be in the UNLOCKED state. The process does not write the lock while spinning, but it only reads it locally, so as the cache coherence traffic reduces. This implements a back off mechanism, where a process that found the lock in the LOCKED state will wait some time before checking it again.

Implementing a synchronization with locks is not so easy. Sometimes processes may need to acquire more than one lock in order to access several memory locations in parallel. That scheme is known as fine grained synchronization and a problematic situation in that scheme is deadlock. Deadlock is a situation in which two or more competing processes are each waiting for the other to finish. For example, there are two threads which hold a lock each and each thread tries to acquire the lock held by the other. As a result, none of the two threads has progress and the execution reaches a dead end.

Atomic operations

A problem with mutual exclusion is that if a thread holding a lock is suspended, all other threads are blocked until the suspended thread resumes, as mutual exclusion is a blocking mechanism and is used in blocking algorithms. In order to avoid this problem there are non-blocking algorithms which use atomic operations instead of locking for synchronization. During an atomic operation a processor can simultaneously read a memory location and write on it in the same bus operation. This prevents any other processor from writing or reading memory until the operation is complete. When a process performs an atomic operation the other processes see it as happening instantaneously.

Atomicity implies indivisibility and irreducibility, so an atomic operation must be performed entirely or not performed at all. Moreover, atomicity is a guarantee of isolation from concurrent processes. The system behaves as if each operation occurred instantly. The advantage of atomic operations is that they are quicker that locks, and do not suffer from deadlock and convoying, as they constitute a non-blocking scheme. The disadvantage is that they only perform a limited set of operations, and often there are not enough to synthesize more complicated operations efficiently. However, the programmer should not reject an opportunity to use an atomic operation in place of mutual exclusion and locks.

Correctness of non-blocking algorithms is challenging to prove and it can be done using linearization points. All functions calls have a linearization point at some instant between their invocation and their response. The state of the shared resource in parallel non-blocking algorithms depends from the order by which the linearization points are reached.

Compare And Swap (CAS) is a fundamental atomic operation used to many non-blocking algorithms. It compares the contents of a memory address to a given value and, only if they are the same, changes the contents of that memory address to the given new value. If the value is up-to-date the operation is successful. If not, the value is stale and has been updated in the meantime by another process, so the operation will fail and the current process must restart the operation. To indicate if the value changed, the return result from

CAS is either a simple boolean value or the value read from that memory address (not the value written to it).

Transactional memory

Another non-blocking scheme for synchronization is transactional memory. Transactional memory is a technology of concurrent processes synchronization that simplifies the parallel programming by extracting instruction groups to atomic transactions. The main benefits are that there are not locks and deadlocks, the parallelism level is increased, so performance is boosted as well and it is relatively easy in use for programming. This synchronization scheme will be further analyzed in next chapter.

Chapter 2

Concurrent Search Trees

2.1 Concurrent Data Structures

A data structure is a particular way of storing and organizing data in a computer, such that they can be used efficiently. Data structures provide a means to manage large amounts of data efficiently and they are used widely in large databases and internet indexing services. Different kinds of applications demand different kinds of data structures and as projects grow larger, it is vital the use of more sophisticated data structures. In fact, the overall performance of the application is limited by the performance of the underlying data structure. As a result, using efficient data structures is the key to design efficient algorithms.

As multiprocessor computer architecture became the dominant computing platform, these data structures had to be redesigned in order to provide safe and synchronized access to multiple threads (or processes). Multiple threads can access data simultaneously, because they run on different processors that communicate with one another. Thus, parallel programming introduces many difficulties and concurrent data structures are far more difficult to design than sequential ones, because threads executing concurrently may interleave their steps in many ways. This requires developers to understand new design methodologies. Furthermore, concurrent data structures have to ensure consistency against the effects of any operation and provide safety and liveness properties. Safety properties usually state that something bad never happens, while liveness properties state that something good keeps happening and the data structure keeps progressing and serving requests.

Designing concurrent data structures for multiprocessor systems also provides numerous challenges with respect to performance and scalability. According to Amdal's law, explained in previous chapter, the sequentially executed parts of the code constitute the most important restricting factor to achieve the maximum gain from parallelization. The operations on a shared data structure belong to that sequential parts of the code. In addition, concurrent data structures are also a restricting factor of application's scalability. It is vital that the speedup of data structures have to grow while the number of processors increases. These data structures are called scalable. In designing scalable data structures,

developers must take care that naive approaches to synchronization can severely undermine scalability.

A second problem in parallel programming and concurrent data structures is memory contention. The results of multiple threads executed in different processors, which demand access to the same locations in memory (same shared data structure), are the overhead in cache coherence traffic and the bus congestion and these two constitute also a restriction in application's performance and scalability. Finally, the attempt to reduce the serial parts of a concurrent data structure and increase the work done in parallel, results to synchronization costs among threads.

2.2 Search Trees

A search tree is a data structure for locating specific values from within a set. It consists of a set of key-value pairs and an interface for accessing and manipulating them. The three main operations of this interface are a lookup operation and two update operations (one to insert and one to delete), as shown in figure 2.1. Various search tree data structures exist, several of which also allow efficient insertion and deletion of elements. In order to reduce search time, search trees have to be reasonably balanced (all the leaves are of comparable depths). In this chapter, we will study three different types of search trees according to their balance: binary search trees, AVL trees and Red-Black trees.

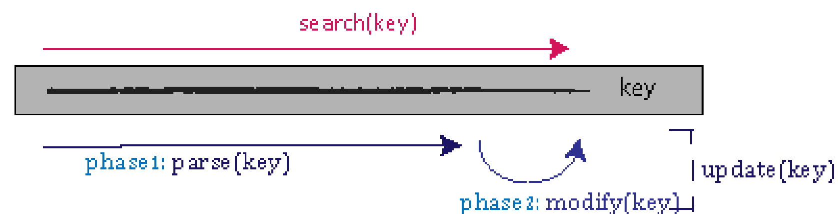


Figure 2.1: Search data structure interface. Updates have two phases: a parse phase, followed by a modification phase.

Search trees are also divided in two categories according to the underlying organization of the key-value pairs in the nodes of the tree. Nodes that have two children are called internal nodes and those that have no children are called external nodes. When the key-value pairs (the useful information) of a search tree are stored only in external nodes (leaves) and the internal nodes of the tree are routing nodes used only as a help for the path to the final external node searched, the search tree is called external tree. When the internal nodes are not only routing nodes, but they also store key-value pairs and are "true" nodes of information, the search tree is called internal tree. The figure 2.2 depicts an external and an internal search tree. We use square shapes to distinguish the leaves, which contain the

key-value pairs, from the internal nodes in the external tree. All the other nodes contain only keys and are used for routing to the appropriate leaf.

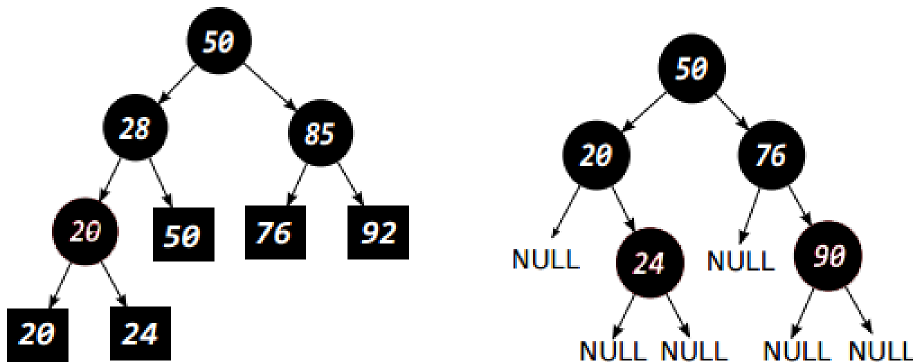


Figure 2.2: An example of a tree in external and internal format.

2.2.1 Binary Search Trees

A binary search tree (BST), also known as an ordered or sorted binary tree, is a node-based data structure in which each node has no more than two child nodes. There is not balance in binary search trees. Each child must either be a leaf node or the root of another binary search tree. Each internal node in BST store a key (and optionally an associated value) and have two distinguished subtrees, commonly denoted left and right. The tree satisfies the binary search property, which states that the key for each node must be greater than all keys in subtree on the left and smaller than any keys in subtree on the right. Duplicate keys are not allowed.

The basic idea behind this structure is to have a storing repository such that the related sorting, searching and retrieving algorithms can be very efficient. Binary search trees are also easy to code and can implement more abstract data structures like dynamic sets of items, multisets, associative arrays and lookup tables that allow finding an item by its key. They have to keep their keys in sorted order, so that lookup and other operations can use the binary search property. While searching for a key in a tree, the traversal begins from the root of the tree to a leaf, the desired key is compared to the keys in BST and deciding, based on the comparison to continue searching in the left or right subtree. If the key is found in BST, the associated value is retrieved. Basic operations (lookup, insertion, deletion) on a BST take time proportional to the height of the tree. On average, each comparison allows the operations to skip about half of the tree, so such operations run in time proportional to the logarithm of the number of nodes n in the tree ($\mathcal{O}(\log n)$). However, in the worst case, where the tree is a linear chain of n nodes, the same operation takes time $\mathcal{O}(n)$.

Although the BST allows fast lookup, addition and removal of items, it has some disadvantages. First of all, the shape of a binary search tree totally depends on the order

of insertions and deletions and can become degenerate (e.g a linked list). In this case, the basic operations take a linear time to the number of nodes in the tree. Moreover, when searching a key in a BST, the key of each visited node has to be compared with the key of the element to be searched. And finally, after a long sequence of random insertions and deletions, the expected height of the BST approaches square root of the number of nodes n, $\sqrt{n}$, which grows much faster than $\log n$.

2.2.2 AVL Trees

The AVL tree is a height balanced binary search tree named after its two inventors, Georgy Adelson-Velsky and Evgenii Landis, who published it in their paper "An algorithm for the organization of information" [3]. It was the first dynamically balanced tree to be proposed. An AVL tree is not perfectly balanced and has two properties. The first AVL property is that every subtree of a node is an AVL tree, too. Assuming that the height of a tree is the number of nodes on the longest path from the root of the tree to a leaf, the second property states that the heights of the two child subtrees of any node differ by at most one. This difference is called **balance factor**. If at any time the balance factor is more than one, rebalancing is required to restore this property. This rebalancing may require the tree to be rebalanced by one or more rotations. Basic operations (lookup, insertion and deletion) take time proportional to the logarithm of the number of nodes n in the AVL tree ($\mathcal{O}(\log n)$) in both average and worst case.

There are many arguments for using AVL trees. First, all the basic operations take time $\mathcal{O}(\log n)$ in the worst case, as the AVL tree is always balanced. Furthermore, the height balancing for the tree adds no more than a constant factor to the speed insertion. And finally, an AVL can never be degenerated in comparison with the BST tree that can be degenerated to a linked list. However, there are also some arguments against using AVL trees. An AVL node has to store the balance factor in order to check violations of the AVL property during insertion or deletion, so an AVL tree demands more space in memory. Programming and debugging an AVL tree is more difficult because of the extra work needed about checking the AVL property and performing rotations. And lastly, although an AVL tree is asymptotically faster than other simple trees, the rebalancing costs time.

Rebalancing in AVL trees

When a thread performs an update operation (insertion, deletion) in an AVL tree, a rebalancing is needed. The rebalancing can be checked through the balance factor of the node. If the balance factor becomes less than -1 or greater than $+1$, the subtree rooted at this node is unbalanced, and a rebalancing is needed. Thus, we have to perform left or right rotations. Figure 2.3 shows a right and a left rotation in a tree without violating the binary search property. An update operation has two phases. The first phase is the standard BST operation (as if the tree were an ordinary binary search tree) and the second includes rotations for rebalancing the tree. When the standard BST operation is performed, there

can be four possible cases of the tree that need to be handled. These are depicted in figures 2.4, 2.5, 2.6 and 2.7. The circles represent the nodes being rebalanced. The triangles T1, T2, T3 and T4 represent subtrees which are themselves balanced AVL trees. The left left case and the right right case are symmetric and need only a single rotation (a right rotation in left left case and a left rotation in the right right case). Similarly, the left right case is symmentric with the right left case and demand two rotations as shown in figures 2.6 and 2.7, respectively.

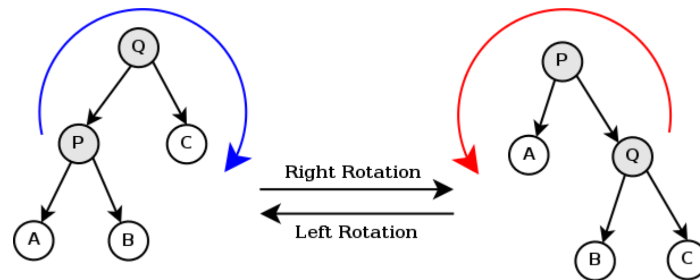


Figure 2.3: A right and a left tree rotation.

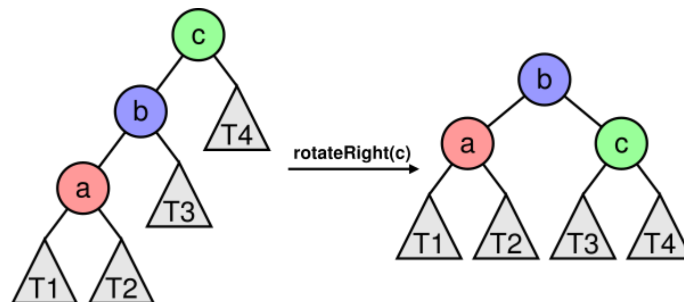


Figure 2.4: Left left case. T1, T2, T3 and T4 represent subtrees which are themselves balanced AVL trees.

2.2.3 Red-Black Trees

A Red-Black tree is also a height balanced binary search tree. The Red-Black tree is derived from the symmetric binary B-tree as described in the paper entitled "A Dichromatic Framework for Balanced Trees" [4]. Each node of a Red-Black tree has an extra bit that represents the color of the node. It can be red or black. This painting of each node of the tree preserves the balance of the tree, that is not perfect, as the Red-Black tree is roughly height balanced. Thus, the Red-Black tree satisfies certain properties named coloring properties, which ensure that the tree remains approximately balanced. When the

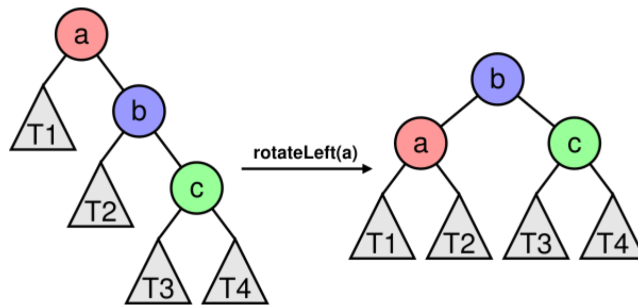


Figure 2.5: Right right case. T1, T2, T3 and T4 represent subtrees which are themselves balanced AVL trees.

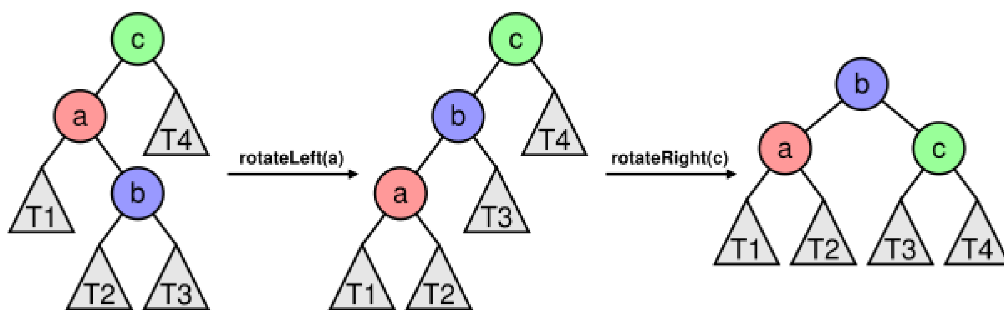


Figure 2.6: Left right case. T1, T2, T3 and T4 represent subtrees which are themselves balanced AVL trees.

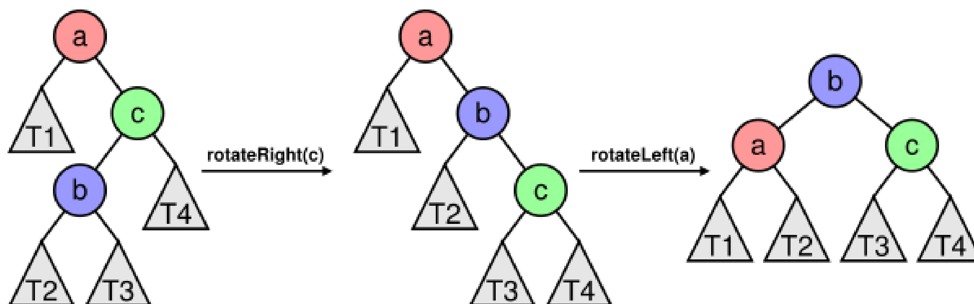


Figure 2.7: Right left case. T1, T2, T3 and T4 represent subtrees which are themselves balanced AVL trees.

tree is modified during operations like insertion or deletion, the tree have to be rearranged and repainted to restore the coloring properties.

A Red-Black tree must satisfy the following coloring properties:

1. A node is either red or black.
2. The root is black (root property).
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black (red property).
5. Every path from a node to a descendent leaf has the same number of black nodes (black property). The number of black nodes from the root to a node is the node's black depth and the uniform number of black nodes in all paths from root to the leaves is called the black-height of the Red–Black tree.

These properties are designed in such a way that the rearranging and the recoloring of the tree can be performed efficiently. Figure 2.8 shows an example of a Red-Black tree. Usually, the leaves of a tree are sentinel nodes as a convenient means of flagging a leaf node and they are black nodes because of the third coloring property. Furthermore, these properties pose another constraint for Red-Black trees: the deepest path in the tree is not longer than twice the shortest one, since all maximal paths have the same number of black nodes according to the last coloring property. More specifically, let B be the number of black nodes of the shortest possible path from the root of the tree to a leaf. The fourth coloring property makes it impossible to insert more than one consecutive red node. Therefore, the longest possible path consists of $2 * B$ nodes, alternating black and red in worst case. Counting the black NIL leaves, the longest possible path consists of $2 * B - 1$ nodes.

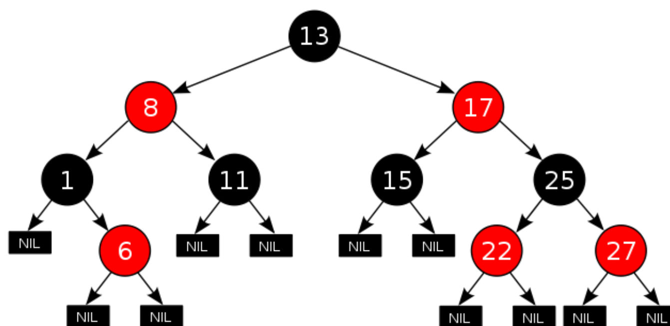


Figure 2.8: An example of a red–black tree.

When inserting or deleting a node, some of the aforementioned properties might be violated and actions must be taken to restore them and rebalance the tree. The two possible violations are: a) *red-red violation*, when a red node acquires a red child (violation of the

fourth coloring property) and b) *double black violation*, when a path of a tree contains one less black node than other paths (violation of the fifth coloring property). To deal with these violations a number of node recolors and rotations are applied.

The basic operations (lookup, insertion, deletion) require worst-case time proportional to the height of the tree $\mathcal{O}(\log n)$, where n is the total number of the nodes in the tree. This theoretical upper bound on the height allows Red-Black tree to be efficient in the worst case, unlike ordinary binary search trees. For example, inserting a key in a non-empty Red-Black tree has three steps. In the first step, the BST insert operation is performed, which takes $\mathcal{O}(\log n)$ time, because the tree is balanced. The second step is to color the new node red, which takes $\mathcal{O}(1)$ time, since it just requires setting the value of one node's color field. And in the third step, a restoration of any violated coloring properties is performed. Restoring these properties requires a small number of color changes and no more than three tree rotations (two for insertion). Changing the color of nodes during recoloring is $\mathcal{O}(1)$. However, it might be need to handle a double-red situation further up the path from the added node to the root. In the worst-case, the fixing of a double-red situation along the entire path from the added node to the root is performed. Therefore, in the worst-case, the recoloring that is done during insert is $\mathcal{O}(\log n)$ (= time for one recoloring * max number of recolorings done = $\mathcal{O}(1) * \mathcal{O}(\log n)$). So overall the third step (restoration of coloring properties) is $\mathcal{O}(\log n)$ and the total time for insert is also $\mathcal{O}(\log n)$.

AVL trees are often compared with Red-Black trees because both support the same set of operations and take $\mathcal{O}(\log n)$ time for the basic operations. A Red-Black tree demands less memory space than an AVL tree, since it requires only one bit of information per node for the color, while an AVL tree demands an integer per node for the balance factor. The Red-Black tree does not contain any other specific data, so in many cases the additional bit of information has no additional memory cost and the memory of a Red-Black tree is almost identical to a classic BST tree.

However, AVL trees are more rigidly balanced than Red-Black trees. The height of an AVL tree is bounded by roughly $1.44 * \log_2 n$, while the height of a Red-Black tree may be up to $2 * \log_2 n$. Thus, lookup is slightly slower on the average in Red-Black trees. On the other hand, the AVL trees may cause more rotations during insertions and deletions. So if an application involves many frequent insertions and deletions, the Red-Black trees should be preferred. And if the insertions and deletions are less frequent and lookup is more frequent, then an AVL tree should be preferred. Red-Black trees can be used in data structures for computational geometry and are valuable in time-sensitive applications such as real-time applications and in functional programming to construct associative arrays and sets, which can retain previous versions after mutations, while AVL trees are attractive for data structures that may be built once and loaded without reconstruction, such as language dictionaries.

Rebalancing in Red-Black trees

When a thread inserts a node, it must be colored red. If the new node is black, then inserting it into the tree always introduces a double black violation. The rest of the algorithm

would then need to concentrate on fixing the double black violation without introducing a red-red violation. On the other hand, if the new node is red, there is a chance that it could introduce a red-red violation. The rest of the algorithm would then need to work toward fixing the red-red violation without introducing a double black violation. However, if the new node is red, and it is inserted as the child of a black node, then no violations occur at all, whereas if the new node is black, a double black violation always occurs. Therefore, the logical choice is to color the new node red, because there is a possibility that insertion will not violate the rules at all. Red-red violations are also more localized and thus, easier to fix.

Similarly with AVL trees, when a thread runs update operations, the tree is modified and the result may violate the coloring properties. To restore these properties, we have to change the colors of some of the nodes (recoloring) in the tree and perform left and right rotations as appearing in figure 2.3. Upon an insertion of a new red node, there are three possible cases of the tree than need to be handled according to [5]:

- Case 1: the uncle of the inserted node is red
  Figure 2.9 shows the tree in case 1. Both the parent and the uncle of the inserted node are red and their parent has to be black. We can fix the problem by flipping their colors (the parent and the uncle of the inserted node becomes black and the grandparent red). However, if the grandparent's color changes, we risk a violation further up the tree. It is possible that its parent could also be red and there is another red-red violation. Therefore, after this case we have to move up the tree and repeat checking for violations.

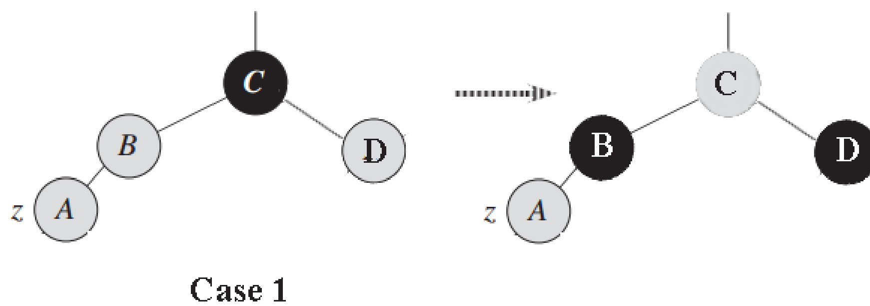


Figure 2.9: Case 1 upon an insetion of a node z in a Red-Black tree. The code for case 1 changes the colors of some node to preserve the coloring properties.

- Case 2: the uncle of the inserted node is black and the inserted node is a left child
  Case 2 is appeared in figure 2.10 In this case, we make the grandparent of the inserted node red and the parent black and we single rotate around grandparent node to the right. This fixes the red-red violation. Nevertheless, the root of this subtree

does not change color. The new root (parent of the inserted node) is black as the previous root (grandparent). In this case, we can be sure that the red-red violation will not propagate upward. Moreover, this rotation does not change the black height of either subtree, so the tree is now balanced and no other rotations or recolors are needed.

- Case 3: the uncle of the inserted node is black and the inserted node is a right child
  Case 3 is also appeared in figure 2.10. In this case a double rotation is needed. We first use a left rotation and the tree becomes the same as in case 2. So, we perform the recolors and a right rotation as they are described in case 2.

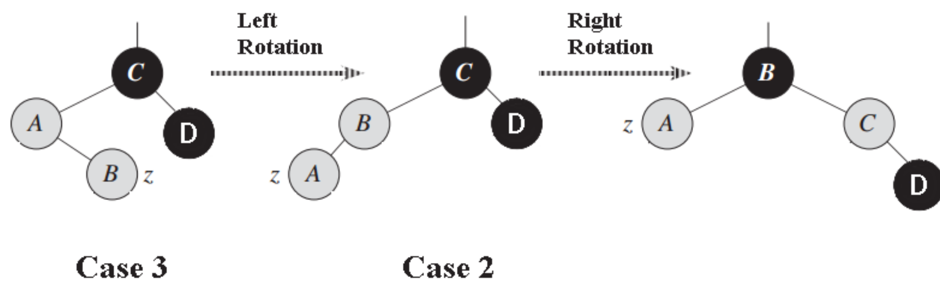


Figure 2.10: Case 2 and case 3 upon an insetion of a node z in a Red-Black tree. We transform case 3 into case 2 by a left rotation. Case 2 causes some color changes and a right rotation to preserve the coloring properties.

The deletion of a node in a Red-Black tree is sure to cause a double black violation, if the deleted node is black. Removing a red node cannot violate any of the coloring properties. Therefore, if we could guarantee that the node to be deleted was red, the deletion would be simplified. When we want to delete node z and z has fewer than two children, then z is removed from the tree, and we want y to be z. When z has two children, then y should be z's successor, and y moves into z's position in the tree. We also remember y's color before it is removed from or moved within the tree, and we keep track of the node x that moves into y's original position in the tree, because node x might also cause violations of the coloring properties. We check color of sibling node to decide the appropriate case. There are four cases to be handled [5]:

- Case 1: x's sibling is red
  Case 1 occurs when the sibling node of x is red (Figure 2.11(a)). Since the sibling node has black children, we can switch the colors of the sibling node and the parent of the node x and then perform a left rotation rooted at the parent of node x without violating any of the coloring properties. The new sibling of x, which is one of the previous sibling's children, is now black and thus case 1 has be converted into case 2, 3, or 4.

- Case 2: x's sibling is black and both of its children are black
  In case 2 (Figoure 2.11 (b)) the sibling node is black as well as its children. We recolor the sibling node red. To compensate for removing one black node, we would like an extra black node to the subtree rooted at x. We fix this by repeating the loop for violations and seting the parent of the x as the new node x.

- Case 3: x's sibling is black, sibling's left child is red and sibling's right child is black
  In case 3 (Figure 2.11 (c)) we can switch the colors of the sibling and its left child and perform a right rotation rooted at the sibling node without violating any of the coloring properties. The new sibling node of x is now black with a red right child, and thus case 3 was converted into case 4.

- Case 4: x's sibling is black and its right child is red
  In case 4 (Figure 2.11 (d)) we color the parent of x and the sibling's right child black as well as the sibling node red and then perform a left rotation. Thus, we can remove the black node x without violating any of the coloring properties.

2.3 Techniques for constructing concurrent data structures

As mentioned in the previous chapter, synchronization is vital in parallel programming such that to ensure that two or more parallel tasks like processes or threads do not simultaneously execute the serial segment of the program (critical section). Furthermore, multiple operations are performed simultaneously in concurrent data structures like concurrent search trees. So in order to ensure that correct results are generated in data structures during multiple operations performed and maintain them consistent, a synchronization mechanism is needed. Synchronization techniques like mutual exclusion and atomic operations are performed in the basic operations of concurrent data structures. Programmers' aim is to construct consistent concurrent data structures which result to high performance and scalability in multiprocessor systems. There are at least three common techniques used today to construct concurrent search trees: coarse-grained locking, fine-grained locking and lock-free programming.

2.3.1 Coarse-grained locking

Coarse-grain locking is a technique to construct concurrent data structures using mutual exclusion and locks. An important property of a lock is lock granularity, which is defined as a measure of the amount of data the lock is protecting. There also two useful concepts related with locks, lock overhead and lock contention. Lock overhead is the extra resources for using locks, like the CPU time for lock initialization and destruction, the memory space allocated for locks, and the time for acquiring or releasing locks. An increased usage of locks in a program results to more lock overhead. And lock contention

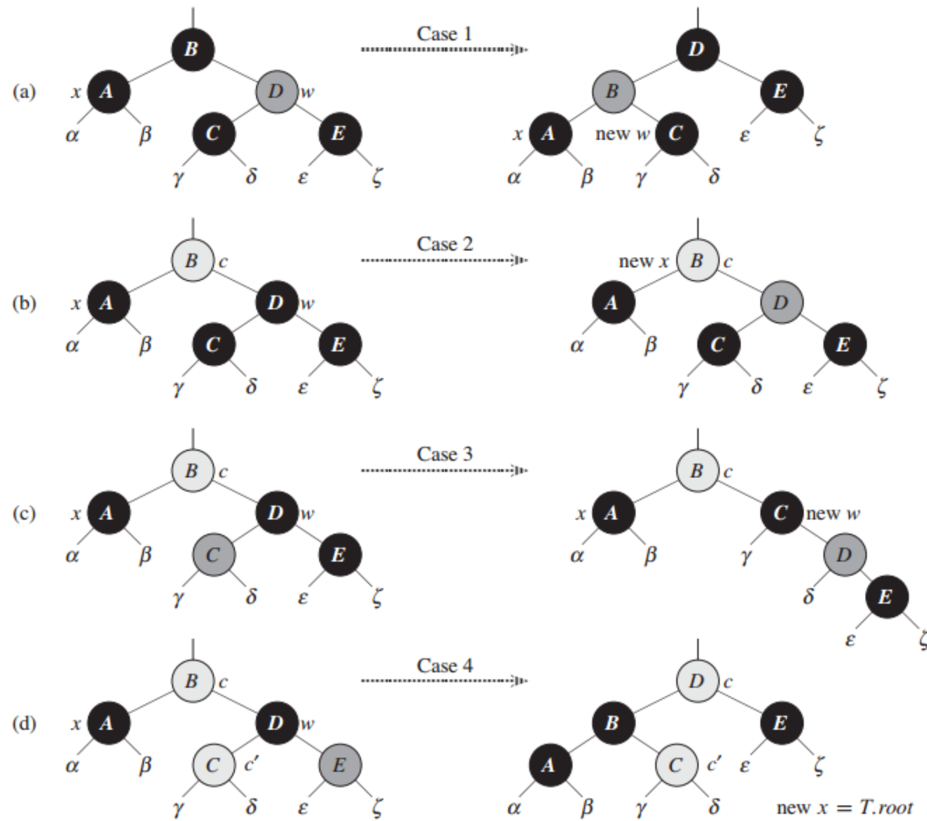


Figure 2.11: The cases upon a deletion of a node in a Red-Black tree. Darkened nodes have color attributes black, heavily shaded nodes have color attributes red, and lightly shaded nodes have color attributes represented by c and c', which may be either red or black . The letters α, β, ..., ζ represent arbitrary subtrees. Each case transforms the configuration on the left into the configuration on the right by changing some colors and/or performing a rotation. Any node pointed to by x has an extra black and is either doubly black (when a black node is deleted and replaced by a black child, the child is marked as doubly black) or red-and-black. (a) Case 1 is transformed to case 2, 3, or 4 by exchanging the colors of nodes B and D and performing a left rotation. (b) In case 2, the extra black represented by the pointer x moves up the tree by coloring node D red and setting x to point to node B. (c) Case 3 is transformed to case 4 by exchanging the colors of nodes C and D and performing a right rotation. (d) Case 4 removes the extra black represented by x by changing some colors and performing a left rotation (without violating the coloring properties). This figure has been taken from [5].

appears when a parallel task like a thread attempts to acquire a lock held by another thread. The less granularity the available locks have, the less likely one process/thread will request a lock held by the other.

In coarse-grained locking, when a process/thread needs to access some shared data, the entire shared data are locked via a global lock, read/write operations are performed on them and then the lock is released. The access to the shared data is serialized and only one process/thread can access them, so concurrency is low. Coarse grained locking is relatively simple to implement, easier to use, understand and debug. The only disadvantage is that it is slow and limits the performance in a multiprocessor system. If there are many threads that need access to the shared data, they will have to wait until the thread that holds the global lock finishes its work and releases the lock. Such a situation means a high lock contention and it degrades the performance. Coarse-grained locking is only useful when the threads execute quickly and do not create a lot of lock contention. On the other hand, it results in less lock overhead when a single process/thread is accessing the shared data.

2.3.2 Fine-grained locking

With fine-grained locking, multiple locks of small granularity are used to protect the smallest possible part of the data structure that the current process/thread needs to operate on. This results to an increased lock overhead because more locks are used for the same shared data, more memory allocation is needed and it appears an additional cost of acquiring/releasing locks. On the other hand, fine-grained locking allows high concurrency and exposes more parallelism by reducing the lock contention for the shared data structure. Multiple processes/threads can proceed in parallel when they do not access the same parts of the shared data structure and it can be good for scalability. Although this technique provides more parallelism, it is complex, much more difficult to implement, as it can be very hard to know which locks are needed and in which order, and can create lock dependencies causing problems like race conditions, deadlocks, livelocks. In order to avoid such problems all processes/threads, that perform operations simultaneously on the data structure, have to acquire locks in the same direction (global order).

So overall lock-based mutual exclusions have many disadvantages like high lock contention and lock overhead and the programmer has to find a solution that trades off some parallelism for reduced overhead. Other disadvantages are priority inversion, where a low-priority process/thread holding a lock can prevent high-priority processes/threads from proceeding, and convoying, where all processes/threads have to wait if a process/thread holding a lock is descheduled. Moreover, the debugging is also a challenge, as bugs associated with locks such as deadlocks are time dependent and extremely hard to identify.

2.3.3 Lock-free programming

Lock-free programming is programming without locks. A lock-free program can never be stalled entirely by any single process/thread and can make progress even if individual processes/threads are suspended indefinitely. Lock-free programming can im-

prove system throughput and robustness (by avoiding situations like the failure of a process/thread holding a lock can lead to a system failure) and has desirable liveness properties. The key in lock-free programming is the hardware support. However, it is very hard to design and implement lock-free algorithms properly and programmers choose to design concurrent data structures using non blocking synchronization which is a portable solution and can be used in different kinds of applications. These data structures are called non-blocking data structures.

Non-blocking data structures can be wait-free, if every operation is guaranteed to be finished in a finite number of steps, lock-free, if some operations are guaranteed to be finished in a finite number of steps and obstruction-free, if an operation is guaranteed to be finished in a finite number of steps, unless another operation interferes. Non-blocking data structures do not rely on locks and mutexes to ensure thread-safety, but on techniques like atomic operations and memory barriers. This means that any process/thread either sees the state before or after the operation, but no intermediate state can be observed (atomicity). Most common atomic operations with hardware support in most multiprocessor architectures are compare-and-swap (CAS), test-and-set (TAS), test-and-test-and-set (TTAS), load-linked/store-conditional (LL/SC).

2.4 Basic Interface in Concurrent Search Trees

As mentioned above, a concurrent search tree consists of a set of elements and an interface to access them. This interface includes three main operations: lookup, insetion and deletion. Each element of the set is stored in a node of the tree and consists of a key-value pair. The key uniquely identifies the element in the set. The three main operations have the following semantics:

- **lookup(key)**: searches for a node containing the given key. If it is found, the value that is bound to the key is returned from the operation, otherwise the operation returns NULL.

- **insertion(key, value)**: attempts to insert a new node in the search tree, binding the given key to the given value. The insertion is successful if there is no other node with the same key.

- **deletion(key)**: attempts to delete the node containing the specific key from the tree. The operation is successful if there such a node exists.

The last two operations (insertion and deletion) comprise two distinct phases. First, there is a traversal in the tree until the desired node is reached (always a leaf in case of insertion) and then the actual modification is attempted.

2.5 A naive approach

In this section, we will present some implementations of concurrent search trees which constitute a naive approach for this data structure. Some of these were developed within the context of the papers [6], [7]. Before describing some implementation details we will introduce two concepts. The first that has already mentioned, is a distinction in search trees depending on the way the key-value pairs are being stored in the tree structure. Internal trees store a key-value pair in every node of the tree and external trees store the values only in the leaves while the internal nodes contain only keys and are used solely for routing purposes. The second concept to introduce is about the order in which the necessary modifications for balancing and the update operation (insertion, deletion) related with the given key are performed. Insertion and deletions consist of two phases. The first one traverses the tree in a top-down manner, i.e from the root towards to the leaves for external trees or towards to the appropriate internal node for internal trees, and locate the place where the node with the key is going to be inserted or the node that is going to be removed from the tree. The second phase is performed only in AVL and Red-Black trees. It traverses the tree in a bottom-up manner, i.e from the lead towards the root, modifying parts of the tree and rebalancing the tree in order to restore the tree properties. Whereas the top-down phase always reaches a leaf, the bottom-up phase backtracks a number of times depending on the violation. In the worst case it shall reach the root of the tree. Thus, when the update operations have these two phases, the implementation is called bottom-up.

In serial implementations, bottom-up trees are very efficient, but in concurrent implementations parallel processes/threads might traverse the tree in opposite directions and in case of fine-grained locking a bottom-up implementation is very complicated. Parallel processes/threads acquire locks in their way and there is no global order for the locks. This may lead to a deadlock. To enable fine-grained synchronization, top-down approaches have been proposed [4], [8], where insertion and deletion are performed in a single top-down pass. For balanced trees (AVL, Red-Black trees) in order to achieve this, while traversing the tree from the root to the appropriate leaf, the necessary modifications (i.e recolors in Red-Black trees) and rotations are applied, ensuring that no bottom-up traversal of the tree is required. In this case, in a concurrent execution, all processes/threads acquire locks in the same direction usually using the well known hand-over-hand technique [9] and avoiding the possibility of deadlock. At each step the nodes that are locked, are released only after the next nodes, that appear lower in the tree are locked. However, as top-down implementations perform generally more tree modifications compared to bottom-up implementations, they impose more overhead and result to worse performance in serial executions.

The lookup operation is a bit more simple and is the same for all three types of search trees (BST, AVL, Red-Black tree). Starting from the root of the tree, a path of nodes is traversed until the node associated with the given key is reached. In a concurrent implementation, in case of fine-grained locking the synchronization in the lookup operations is achieved with the hand-over-hand locking techinque [9], too. Locking is performed at each distinct step of traversal and as in update operations (insertion, deletion), the lock of

the next node to be visited is acquired before the lock of the current node is released.

2.5.1 Description

We have developed nine different implementations of concurrent search trees, internal and external trees, using the aforementioned techniques for synchronization and both bottom-up and top-down approaches for rebalancing. Furthermore, there are no duplicates in the trees. The insertion of a key is successful, only if the key does not exist in the tree, and in this case a new node with a key-value pair is inserted, otherwise it returns false as a sign for unsuccessful insertion.

Our concurrent search trees are:

- avl-bu-cg-ext-lock tree
  - AVL tree
  - external tree
  - coarse-grained locking
  - bottom-up approach for rebalance
  - iterative implementation

- bst-td-fg-int-lock tree
  - BST tree
  - internal tree
  - fine-grained locking
  - top-down (no rebalance needed)

- rbt-bu-cg-ext-iter-lock tree
  - Red-Black tree
  - external tree
  - coarse-grained locking
  - bottom-up approach for rebalance
  - iterative implementation

- rbt-bu-cg-int-iter-lock tree
  - Red-Black tree
  - internal tree
  - coarse-grained locking
  - bottom-up approach for rebalance
  - iterative implementation

- rbt-bu-cg-ext-rec-lock tree
  - Red-Black tree
  - external tree
  - coarse-grained locking

  - bottom-up approach for rebalance
  - recursive implementation

- rbt-td-cg-ext-lock tree
  - Red-Black tree
  - external tree
  - coarse grained locking
  - top-down approach for rebalance

- rbt-td-fg-ext-lock tree
  - Red-Black tree
  - external tree
  - fine-grained locking
  - top-down approach for rebalance

- rbt-td-cg-int-lock tree
  - Red-Black tree
  - internal tree
  - coarse grained locking
  - top-down approach for rebalance

- rbt-td-fg-int-lock tree
  - Red-Black tree
  - internal tree
  - fine-grained locking
  - top-down approach for rebalance

2.5.2 Implementation details

- All these implementations were developed in C programming language.

- In order to implement parallelism we used POSIX threads or Pthreads as a parallel execution model. POSIX threads is an API defined by the standard IEEE POSIX 1003.1c.

- False sharing is the most limiting factor on achieving scalability for parallel threads of execution in a symmetric multiprocessor system (SMP), where each processor has a local cache. It occurs when threads on different processors modify independent variables that share the same cache line. This situation invalidates the cache line and forces an update, which degrades performance. Thus, in order to avoid false sharing and align the node structure of the tree in memory, we applied structure padding. This is a technique in which one or more bytes are inserted between memory addresses. As a result, nodes of the tree that were previously allocated in consecutive addresses in memory, now reside on different cache lines. Listing

2.1 and figure 2.12 show a typical bst node structure and a representation of it in memory, respectively. Supposing that a cache line is 64 bytes, the padding is 64 bytes minus the total 'real' bytes of information of a node, such that each node will be allocated in exactly one cache line.

Listing 2.1: A typical bst node structure

```
1  typedef struct bst_node{
2          int key;
3          struct bst_node *lchild;
4          struct bst_node * rchild;
5          void *value;
6
7          char padding[CACHE_LINE_SIZE - sizeof(int) -
8                  2 * sizeof(struct bst_node *) -
9                  sizeof(void *)];
10 } bst_node_t;
```

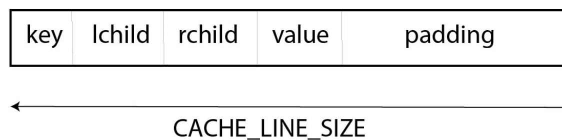


Figure 2.12: A typical bst node structure in memory using padding, such that to be the same bytes as one cache line.

- Iteration is a loop based imperative repetition of a process that repeats some part of the code and recursion a method where the solution to a problem depends on solutions to smaller instances of the same problem. A recursive function calls itself again to repeat some code for a smaller piece of a complicated task and then it combines the results. We have implemented iterative versions of concurrent Red-Black trees as well as a recursive and an iterative version of an AVL tree. In an attempt to implement a bottom-up and iterative approach, we used stack structure. While a thread traverses the tree with a top-down manner until the desired node is reached, we store the path of nodes in the stack such that to access the reverse path with a bottom-up manner and rebalance the tree. As concerning the recursive version, the basic operations of the tree (lookup, insertion, deletion) are recursive functions which call themselves within the program text and when they return, they rebalance the tree if needed. The disadvantage of these recursive functions is that they traverse the whole path from the root to the appropriate for modifications node (always a leaf in external trees) two times, while the iterative, bottom-up approach does not always traverse the whole reverse path to the root in the bottom-up phase, but it stops when no rebalancing is needed.

- In order to implement a coarse-grained locking there is a single global lock (pthread spinlock) shared for all threads. Every thread that performs one of the three main operations waits until the lock is released, acquires the lock, executes the operation and then releases the lock. Only one thread acquires the shared lock at a given time. Coarse-grained locking is easy to implement, but leads to serialization of accesses.

- In fine-grained locking version of concurrent search trees, each node structure has its own lock. Thus, we have added an additional field in the node structure for a pthread spinlock. We only implement top-down approaches because, as explained before, a bottom-up approach is hard, if possible at all, to be implemented. We also keep a global order in locks (acquire locks always in the same direction) to avoid deadlocks. More specifically, while a thread traverses the tree from the root to the leaves, it acquires locks with a top-down manner (lock the node which locates in a higher level of the tree first and then the node in the lower level) via hand-over-hand locking technique [9]. However, internal and external trees entail different requirements and challenges in fine-grained locking. An important one is that in order to delete an internal node (node with two children) from an internal tree, we have first to find its successor (the leaf node with the greatest key that is less than the key of the node to be deleted), swap their key-value pairs and remove the successor leaf node. This operation requires exclusive access to the every node between these two nodes. To achieve this, we keep locked the internal node that we want to delete, until its successor (leaf) node is found. As shown in figure 2.13 the whole subtree rooted at this internal node is locked. If we do not keep locked the internal node, that we want to delete, unlock it and acquire again its lock after we find its successor, it may occur deadlock because of acquiring locks with the opposite direction (first the lower level leaf-successor and then the higher level internal node to be deleted). On the other hand, a deletion in an external tree involves only leaf nodes and there is no such problems.

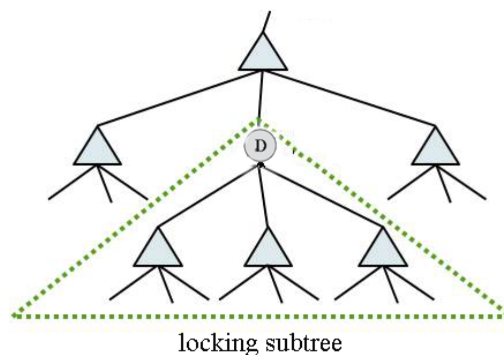


Figure 2.13: Deletion of node D in an internal tree. While searching D's successor to swap their key-value pairs, the subtree rooted at D node must be locked.

- In avl implementation we have also added an additional field in the node structure which represents the height of the node, such that to compute the balance factor and perform rebalancing to restore the AVL property. Similarly, in Red-Black trees each node has a color in order to restore the coloring properties. So, the structure of a Red-Black node has also an additional field for the color.
- Finally, in internal implementations the leaves of the tree are sentinel nodes. Sentinel nodes are designated nodes used as traversal path terminators instead of NULL pointers. Using sentinel nodes, we reduce the code size by avoiding additional checks for NULL pointers.

2.6 A more sophisticated approach

This section includes more sophisticated approaches for concurrent search trees. These implementations are complex algorithms (lock-based or lock-free) taken from the paper [10] and use a more complicated synchronization mechanism than naive concurrent search trees. Thus, we expect to have higher performance and scale better. We will briefly describe the concept of each implementation.

2.6.1 Description

Bronson

This is a concurrent relaxed balance AVL tree proposed at [11], that delivers high performance, good scalability and tolerates contention by controlling the size of critical sections (all updates have fixed size critical sections) and taking advantage of validation logic. A lookup can block until a concurrent update is completed. In an attempt to check concurrent updates, this algorithm uses version numbers. In version number there is a "changing" bit to indicate if a write is in progress and the remainder of the bits form a counter. Each node has a version number, such that to verify if a read is still valid. For example, at time t_1 a thread executes a read and the associated version number is v_1. The thread must block until the change bit in version number is not set (a concurrent write to be completed), read the protected value x and then at time t_2 rereads the version number v_2. If $v_1 = v_2$, then the read is still valid at t_2.

Moreover, as a concurrency control mechanism for searching and traversing the binary search tree the algorithm performs hand-over-hand optimistic validation. Hand-over-hand locking [9] reduces the duration over which locks are acquired by releasing locks on nodes whose rotation no longer affect the correctness of the search and optimistic validation is used to protect critical sections and is chained with hand-over-hand approach. If a key is present in the set of elements, then a thread must traverse the tree from the root to the node associated with the key. Similarly, if a key k is not present in the tree, then a thread must reach the node that would be the k's parent, if it were inserted. Through hand-over-hand optimistic validation, in a lookup, which consists of an interval of keys, the algorithm

attempts to check whether a key is absent from the entire tree or present in the current subtree. Each time a lookup operation navigates downward to the tree after performing a comparison between keys the interval is decreased. At all times the interval includes the target key, so if the subtree ever becomes empty, it means that there is no node with that key present in the entire tree. The optimistic validation scheme only needs to invalidate lookups whose state is no longer valid.

Finally, the described tree is referred as partially external tree. In internal trees with no routing nodes the deletion of a node with two children requires that the node's successor must be unlinked from the tree and linked in node's position in the tree. This unlink and relink of node's successor in concurrent trees must be done atomically and every node along the path from the node to be deleted to its successor must be locked. This excessive locking limits scalability and performance. On the other hand, in external trees there is no such problems in a deletion of a node with two children. However, external trees with N nodes require $N - 1$ routing nodes and it increases the storage overhead and the average search path. As a result, this algorithm uses a simple scheme referred in [11] as partially external tree that simplifies deletions by leaving a routing node in the tree when deleting a node has two children. When rebalancing is performed, routing nodes with fewer than two children are unlinked from the tree. A deletion of a node with fewer than two children is handled immediately unlinking the node. Partially external trees require fewer routing nodes in most cases than an external tree after a sequence of update operations, but in the worst case they may have exactly the same number of routing nodes. As concerning the node structure, this implementation has the same node structure for both key-value associations and routing nodes and this permits a value node to be converted to a routing node (or the reverse) by changing a field in the node structure and without modifying other inter-node links.

Drachsler

This tree presented in [12] is a BST internal tree that uses logical ordering among nodes. The key to design correct and efficient concurrent binary search trees is to implement a scalable design for the lookup operation. The tree ordering layout is separated from the tree physical layout and thus, lookup operations can proceed concurrently with operations that modify the physical tree layout without synchronization. This approach allows fast lookup operations. The authors of [12] exploited this idea to obtain an intuitive, simple and robust lookup operation, that also provide strong progress guarantees.

The logical ordering among elements can be viewed as consecutive intervals. For instance, the logical ordering for the elements $1 < 3 < 5 < 7 < 9$ can be viewed as intervals $(-\infty, 1), (1, 3), (3, 5), (5, 7), (7, 9), (9, +\infty)$. An element belongs to the tree if and only if it is an endpoint of some interval and does not belong otherwise. The algorithm for these concurrent binary search trees uses intervals to answer lookup requests and to synchronize operations. Each node of the tree keeps its successor endpoint (succ field in the node structure) and its predecessor endpoint (pred field in the node structure), which are unique. The synchronization may also be performed on these endpoints. As this tree

is a fine-grained locking approach, two locks are needed, a treeLock, which protects the tree's physical layout fields (left, right, parent), and a succLock which protects the logical layout fields (the succ field and the pred field of the node). That is, for a node n, n's succLock protects the interval $(n, succ(n))$.

A key k is present in the tree if it is the endpoint of an interval ($k \in [k_1, k_2]$). Thus, in a lookup operation, we have to find if there is such an interval. Lookup operation has two phases. Firstly, there is a traversal of the physical tree layout until a leaf is reached and if the key k was found during traversal, then the key k is in the tree. And secondly, if the key k was not found, it must be found an interval with keys k_1 and k_2, that are in the tree and such that $k \in (k1, k2)$. This search for key k terminates when it reaches a node of value $\tilde{}k$ where: (i) $k \in (pred(\tilde{}k), \tilde{}k)$, or (ii) $k \in (\tilde{}k, succ(\tilde{}k)$. This will be done via the logical ordering pointers, the predecessor (pred) and the successor (succ).

Eventually, as in all concurrent search trees a synchronization mechanism is needed. According to [12], each update operation is performed in four steps:

1. Acquire logical ordering layout locks.
2. Acquire physical layout locks.
3. Update the logical ordering layout and release logical ordering locks.
4. Update the physical layout and release physical locks.

As mentioned, acquiring the tree's physical layout locks (treeLocks) prevents simultaneous updates to the node's physical layout information like node's children, parent, e.t.c. and acquiring the tree's logical ordering layout locks (succLocks) prevents simultaneous updates to the intervals. Each interval is associated with a lock, the succLock of the node with the beginning number of the interval as its key. When an operation updates two intervals (like merging two intervals in deletion), it must acquire two succLocks for the two intervals. Therefore, there is synchronization via locks for both the tree physical and logical ordering layout. In an attempt to avoid deadlocks, succLocks, which are used for the tree logical ordering layout should be acquired before treeLocks, which are used for the tree physical layout. Between two succLocks the lock of the node with the smaller key should be acquired first to keep a global order and between two treeLocks the lock of the node that appears lower in the tree should be acquired first. However, deletion operation must acquire treeLocks against the locking order. As a result, when locking treeLocks against the locking order is required, threads optimistically attempt to acquire them without blocking on it (using tryLock()) and if they fail, they release all locks and the operation is restarted. In this way, deadlock cannot occur.

BST Ticket

This concurrent binary search tree proposed in [10] is a lock-based BST implementation. It attempts to reduce the number of acquired lock per update operation and as a consequence the number of cache lines transfers. More specifically, BST Ticket is an external tree, where every internal node used for routing purposes is protected by a lock and contains a version number. As referred in [10] bst-tk stands for BST Ticket and has ticket

locks for locking and keeping track of version numbers of nodes. The version numbers are used in order to be able to optimistically traverse the tree and later detect concurrency. It can be validated in order to avoid concurrent conflicting updates. Based on the observation that a ticket lock already contains a version field, the algorithm integrates the version validation and increment, with locking and unlocking, respectively. The interface of ticket locks is modified so that the lock acquisition involves the version number of the node and as a result performing a locking and validating the version can be done in a single step. Briefly, the lookup operation is executed in a wait-free manner and an update operation traverses the tree until the appropriate for modifications node is found, acquires a number of locks and executes the update. If the lock acquisition fails, the version of the lock has been incremented by another concurrent update and the operation has to be restarted. Furthermore, the tree is optimized by allocating two small ticket locks for each node, such that the left and the right child pointer of a node can be locked separately.

In insertion operation the first phase is a lookup operation that keeps track of the predecessor node apart from the current one. If the update is possible, two nodes are allocated, an external node that stores the key-value pair and a routing node, and then a lock is acquired, protecting either the left or the right child pointer of the predecessor. Once the locking succeed, the update is performed, otherwise the operation is restarted. Similarly, the lookup phase of a deletion operation keeps tracks of both predecessor and the predecessor of the predecessor, as a deletion influences both nodes. If the deletion is possible, the appropriate child pointer (left or right) of the predecessor of the predecessor is locked, as well as both pointers of the predecessor (this is done in a single step). If both acquisitions are successful, then the deletion is performed, otherwise the operation has to be restarted. Overall, this implementation demands one lock for a successful insertion and two locks for a successful deletion.

Aravind

This concurrent binary search tree is presented in the paper [13]. It is a lock-free algorithm that supports the three basic operations for binary search trees. Lock-freedom requires that some process be able to complete its operation in a finite number of steps. Thus, this lock-free approach uses two atomic operations for reads and writes, compare-and-swap (CAS) and bit-test-and-set (BST). As in previous concurrent search tree, an external representation of search trees has been selected. In order to limit the conflicts among update operations and reduce the overhead of update operations there are some optimizations. As mentioned in [13] (i) the algorithm is based on marking edges as deleted rather than nodes, (ii) it does not use explicit objects for coordination between conflict operations and finally, (iii) it permits multiple keys (nodes) being removed from the tree in a single step. As a result, update operations in this algorithm work on a smaller portion of the tree (smaller contention window), allocate fewer objects and execute fewer atomic operations (one for insertion and three for deletion).

In contrast to previous implementations this algorithm marks the edges as deleted instead of the nodes. Each update operation becomes the "owner" of some edges that it

needs to work on. Note that every edge has a tail and a head node. Marking an edge means that either both tail and head nodes or only its tail node will be deleted from the tree. Thus, it is vital to distinguish between these two cases. In paper [13], the first type of marking is referred as *flagging* and the second type as *tagging* and as for the implementation, to enable flagging or tagging they exploit two bits (denoted flag and tag) from each child address (each child pointer in the node structure). If one of the two bits in a child address has value 1, then the corresponding outgoing edge has been marked (flagged or tagged), otherwise the edge is not marked. For example, in a deletion operation of a node (leaf) n, marking an edge means setting the flag bit in the child field of n→parent, that points to n→leaf to 1 (set a bit in the pointer that is associated with the left or the right child). Additionally, as explained in section 3.2.4 [13], if there are multiple edges marked in the tree, this will cause multiple leaf nodes to be removed from the tree in a single step during deletion operation.

Finally, in this algorithm there is also a helping strategy that is performed only in deletion operations. There is no helping strategy for insertions. This is why a helping strategy increases the overhead of an operation and may provoke duplication of work. Moreover, this algorithm does not use explicit objects for coordination, but steals two bits from the child address of the node structure. In an insertion operation of a node n, a helping strategy needs to be performed when it is discovered that the the edge from n→parent to n→leaf exists and has been marked (flagged or tagged). It means that a concurrent deletion operations is attempting to delete n→parent from the tree. As a result, the insertion operations helps the concurrent deletion operation to complete (n→parent and one of its children to be removed from the tree). Subsequently, the insertion operation restarts from the beginning (lookup phase). Similarly, the deletion operation of a leaf node n performs a flagging of the edge from n→parent to n→leaf using CAS atomic operation. If the CAS operation fails, the deletion operations executes a helping strategy in the same way as in insertion operation and then it tries again by retrying the lookup phase.

Ellen

The last implementation is presented in [14] and describes a non-blocking and linearizable binary search tree (BST). This algorithm is a lock-free version of a BST that uses non-blocking synchronization and more specifically the atomic operation compare-and-swap (CAS). Therefore, it can tolerate any number of crash failures. Furthermore, as mentioned in [14] they use a leaf-oriented BST that has already presented as external tree. All keys of the set of elements for the tree are stored in leaves and every internal node that has exactly two children is used to find the path to the correct leaf and its key may or may not be in the set of elements. Update operations (insertions and deletions) that alter different parts of the tree and do not interfere with one another can proceed concurrently. Lookups only perform reads of shared memory and traverse the tree from its root to a leaf (as this is an external tree), so they do not interfere with updates, too.

In update operations the appropriate modifications for the tree are performed using the atomic operation CAS. However, in concurrent updates this can lead to problems and

in an inconsistent state of the tree. To avoid analogous problems there is a mark field in the node structure named "state", so that in a deletion operation of a leaf, the parent's state field is set before unlinking the parent from the tree. Setting the node's state field through CAS steps ensures that its child pointers cannot change. This field is also used to flag the node to indicate that an update is attempting to change a child pointer of the node. Before an update operation, that changes either of node's child pointers, the state field is changed to a flag value according to the update operation (insertion or deletion) to be performed. After the termination of the update operation the state changes back to a "clean" state. Generally, setting the state field of a node is similar to locking its child pointers. In concurrent search trees an operation must successfully acquire the lock in order to alter node's child pointers, because this ensures that they never can change until releasing the lock. A lookup operation does not change any child pointer and thus, it does not acquire any locks. On the other hand, insertion operation is guaranteed to complete when it acquires a lock (sets the state field) of a single node and a deletion operation after acquiring locks of two nodes. Since only update operations need to acquire locks of one or two nodes near a leaf of the tree, this locking scheme does not provoke serious contention problems and concurrent updates which do not interfere with one another (perform modifications on different parts of the tree) can proceed simultaneously.

In this implementation there is also a helping strategy. More specifically, a process helps another process's operation to finish only if the other operation is preventing its own progress. As a lookup operation does not modify the tree and cannot never be blocked, it never helps any other operation. Nevertheless, an update operation that must lock a node (setting the state field in node structure) that is already locked helps complete the operation that locked the node first and then in retries its own update operation. In an attempt to achieve this helping strategy, there is an Info record. When an operation locks a node, it stores enough information (in an Info record), so that another process that requests the locked node, to help complete the operation. This mechanism suffices to achieve a non-blocking synchronization. According to the update operation to be performed there are two types of Info record, as an insertion and a deletion operation demand different information to be stored. As described in [14], to complete an insertion a process must have a pointer to the leaf which is to be replaced, that leaf's parent and the newly created subtree that will be used to replace the leaf. And these are the information to be stored to an Info record for an insertion. Similarly, to complete a deletion operation a process must have a pointer to leaf to be deleted, its parent, its grandparent and a copy of the state and info fields of the parent. So, these information are stored to an Info record in a deletion operation. Therefore, if a insertion finds that some other operation has locked (has set the state field of the node structure) the parent of the node that is to be inserted, it helps the other operation to complete and then retries. If a deletion operation finds that the parent or the grandparent of the node to be deleted has already locked, it helps that operation to complete and then starts over with a new attempt. However, a deletion operation demands two locks, one for node's grandparent (acquiring it first) and one for node's parent. Thus, it is possible that a deletion will fail to complete after the grandparent is locked, because the parent is already locked by another process. In this case, it helps the operation that

locked the parent to complete and performs a backtrack CAS to release the grandparent.

2.6.2 Implementation details

These five implementations were developed in C programming language using POSIX threads within the context of the paper [10]. The code is avialable at http://lpd.epfl.ch/site/ascylib.

2.7 Evaluation

2.7.1 System Configuration

The system we used to evaluate the implementations was a 60-core platform (Figure 2.14), NUMA architecture with the following characteristics.

- 4 sockets (Intel(R) Xeon(R) CPU E7-4880 v2 @ 2.50GHz)
- 15 cores per socket (30 threads with hyperthreading)
- 32KB L1 data cache per core
- 32KB L1 instruction cache per core
- 256KB L2 cache per core
- 38MB L3 cache per socket
- 1TB RAM

2.7.2 Run Configurations

To evaluate the implementations of the described concurrent search trees, we perform random operations varying the number of threads, the range of the set of elements from which the keys are selected and the proportion of lookup, insertion and deletion operations. More specifically:

- Each software thread is manually pinned to a hardware thread in order to take advantage of the locality with the sockets. For instance, in case of a small number of threads we pinned them in the same socket, so as to share the same L3 cache. We also pin software threads to cores in such a way that all the available physical cores are being employed before utilizing hyperthreads. Otherwise, if we did not pin the software threads, the operating system may execute many software threads in the same core leaving another core idle.
- The duration of each execution is 5 sec, during which each thread performs randomly chosen operations, based on the percentage of operations we have selected.

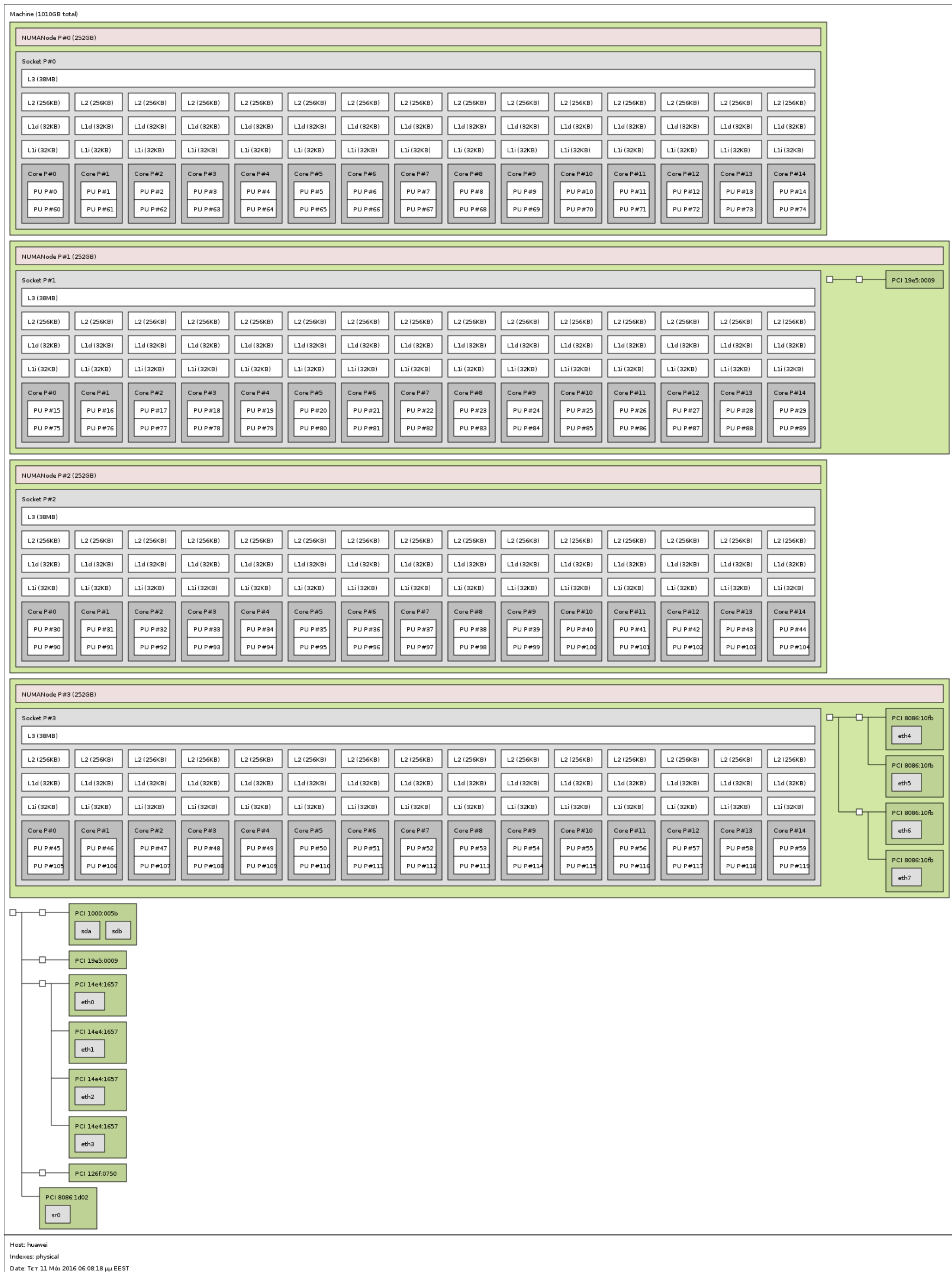


Figure 2.14: The platform used in evaluation of concurrent search trees.

- The key range effectively determines the size of the tree, we evaluate the concurrent search trees for ranges 2K, 32K and 2000000 keys. In the beginning of each execution the tree is initializes with half the possible keys of the selected range. This provides the guarantee that on average half of the operations are successful and that the average execution time of each operation remains the same all over the whole execution (the percentage of insertion operations is the same as deletion operations, so that the tree remains approximately the same).

- We use various proportion of operations, three different workloads 80-10-10 50-25-25 20-40-40, with 80%, 50% and 20% of operations respectively being lookups in the tree, i.e. read-only traversals, while the rest are equally divided between insertions and deletions. These workloads represent a read-dominated, read-write and write-dominated access pattern on the tree respectively.

2.7.3 Results

Naive concurrent search trees

Figures from 2.15 to 2.17 depict the throughput obtained from executions of the naive concurrent search trees on the described 60-core platform. Trees with key range 2K do not scale and in this case the performance is reduced as the number of threads increases. In small trees there are many conflicts on nodes among threads and as a result high contention. While the key range of the tree expands (32K and 2000000), the fine-grained implementations scales until a number of threads and after that point the performance collapses. More specifically, they scale up to 8 threads. After the point of 15 threads, the throughput is decreased and it appears the effect of NUMA architecture as threads employs more than one socket of the platform. In a cache miss during an operation the transfer of a cache line for a node is very expensive from one socket to another. On the other hand, coarse-grained locking versions do not provide parallelism. They are protected by a single global lock and as a result they serialize all accesses on the tree. They used as a baseline.

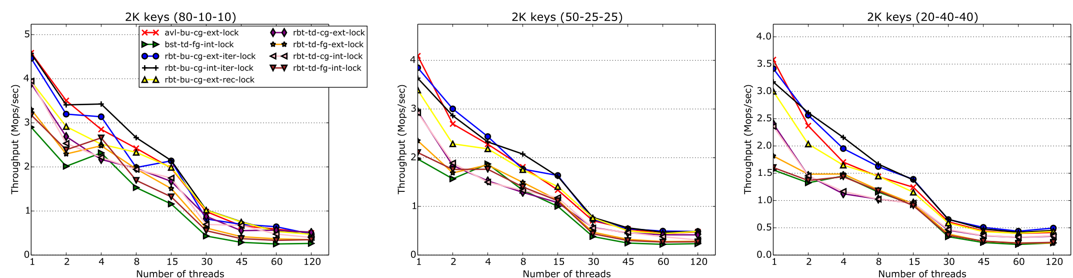


Figure 2.15: Throughput of concurrent naive implementations for 2K key range and the three workloads.

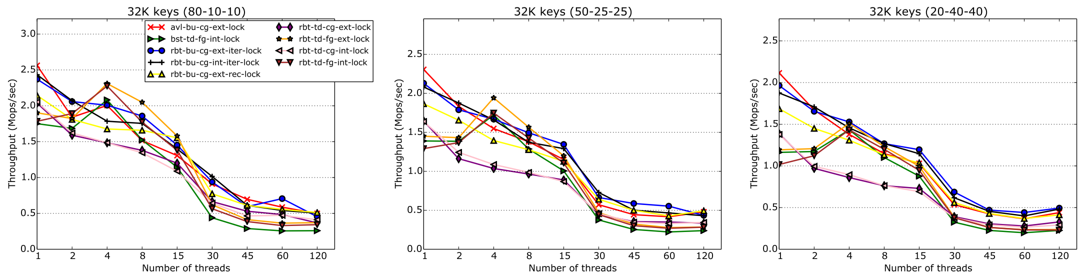


Figure 2.16: Throughput of concurrent naive implementations for 32K key range and the three workloads.

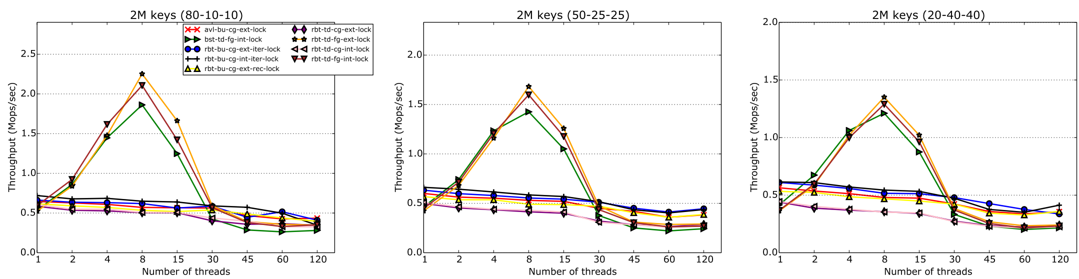


Figure 2.17: Throughput of concurrent naive implementations for 2000000 key range and the three workloads.

Among the fine grained-implementations, the RBT external tree version has the highest performance. This is because it is a height-balanced tree comparing to the BST fine-grained version and performs faster deletions comparing to the RBT internal version. As already mentioned, the deletion operation in an internal fine-grained locking tree requires exclusive access to every node between the node to be deleted and its successor. To achieve this in a fine-grained version the node to be deleted is kept locked until its successor is found. In this way, the whole subtree rooted at this node is locked and no other thread can proceed in it. All threads that attempt to proceed in this subtree block. This results to faster deletions in an external version which involves only leaf nodes. On the other hand, in coarse-grained implementations the internal version has a better throughput than the external version. In internal trees a lookup operation is faster as it can terminate in a small depth of the tree, while in external trees the operation terminates when a leaf is reached.

In top-down approach, while traversing the tree from the root to the appropriate leaf, modifications are proactively performed in order to guarantee that no bottom-up traversal of the tree is required. This pessimistic nature of top-down approach generally results to more tree modifications for each operation compared to bottom-up which performs only the necessary modifications for the operation. Consequently, top-down approaches have worse performance in serial executions (coarse-grained locking) and impose more over-

head. In our results bottom-up coarse-grained locking trees have higher throughput than top-down, as the cost to traverse two times the path to the appropriate node executing only the necessary modifications is lower than traversing it one time, executing modifications in advance to avoid bottom-up traversal (typically more modifications are performed).

Comparing bottom-up iterative and recursive implementations, the iterative bottom-up concurrent search trees perform better than recursive versions. The iterative trees terminate the bottom-up traversal when no other rebalancing is needed (in a balanced node, not necessary the root node), while the recursive trees in bottom-up phase have to return up to the root of the tree. Thus, recursive trees traverse the path to the appropriate node exactly two times performing the necessary modifications. Furthermore, there is usually more overhead associated with the recursive calls due to the fact that the call stack is so heavily used during recursion.

Finally, we can note that in small trees the throughput in coarse-grained locking implementations is reduced significantly as the number of threads increases, in contrast to larger coarse-grained locking trees, where throughput is approximately the same for the different number of threads. This happens because the operations in small trees take a little time. As a result, as the number of threads increases, each thread blocks for a long time and performs a fast operation, while the same operation in a large tree lasts more time. Therefore, the operations performed in the same amount of time in small trees are much less in case of multiple threads than that in case of one thread.

More sophisticated concurrent search trees

The results (Figures 2.18, 2.19, 2.20) of our evaluation are close enough to those presented at [10]. Aside from BST Ticket search tree, Aravind concurrent search tree is generally the best concurrent implementation. This lock-free implementation uses two atomic operations on average per update operation, which is close to a concurrent search tree without any synchronization in terms of the number of stores and the number of the affected cache lines. More specifically, this tree has a small contention window because it operates at edge-level (marking edges as deleted instead of nodes). Secondly, it allocates fewer objects and executes fewer atomic instructions per update operation than the other algorithms. It does not use explicit objects for coordination between conflicting operations (like Info record in Ellen concurrent search tree). Helping strategy is performed only for deletion operations and instead of using explicit objects for coordination, the algorithm uses a small number of bits from child pointers stored in the node structure to enable coordination between operations. There is no helping strategy for insertions because it increases the overhead of an operation and may provoke duplication of work. And as already explained, in this algorithm, multiple leaf nodes may be deleted (unlinked from the tree) in a single step. To sum up, the algorithm reduces the contention between update operations and lowers the overhead of an update operation.

Comparing BST Ticket concurrent search tree with Aravind search tree, they have very similar behavior as they both are close to an asynchronous search tree. BST Ticket tree outperforms lightly the Aravind search tree (Figure 2.18 and 2.19 for smaller trees)

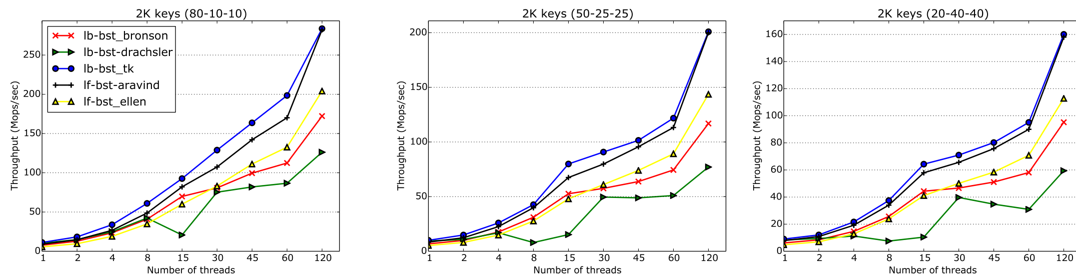


Figure 2.18: Throughput of concurrent sophisticated implementations for 2K key range and the three workloads.

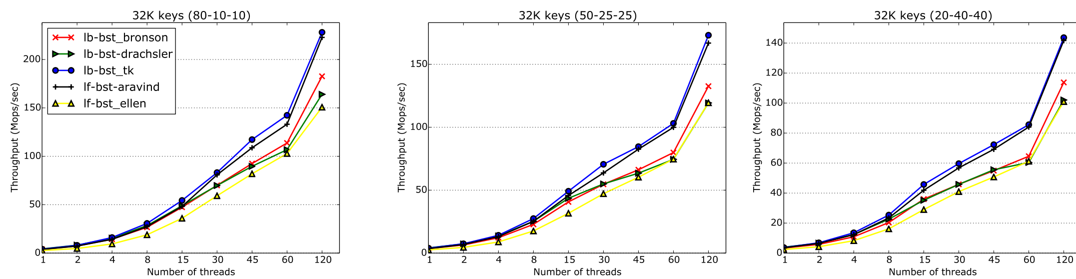


Figure 2.19: Throughput of concurrent sophisticated implementations for 32K key range and the three workloads.

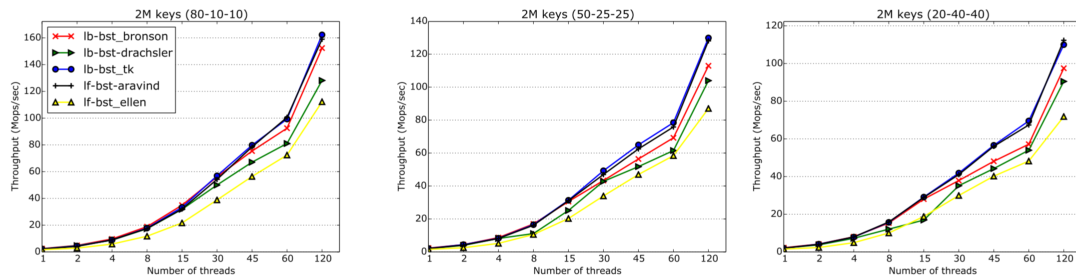


Figure 2.20: Throughput of concurrent sophisticated implementations for 2000000 key range and the three workloads.

as it executes less atomic operations per update. It executes two atomic operations per deletion while Aravind search tree executes three atomic operation per deletion (they both execute one atomic operation per insertion). However, BST Ticket has slightly increased parsing overhead compared to Aravind and in large trees where the contention between threads is not so high, they have approximately the same behavior (Figure 2.20).

The rest three trees (Bronson, Drachsler, Ellen) have worse throughput for all the three key ranges used in our evaluation. They have a more complicated synchronization mecha-

nism. Ellen concurrent search tree employs helping strategy for both insertion and deletion operation and only on elements that the current operation wants to modify. Helping strategy is typically expensive, as it requires additional synchronization to be implemented and imposes additional overhead for each operation. Furthermore, Ellen concurrent search tree uses explicit objects (Info record) that augment the number of stores per operation and the number of cache line transfers. Secondly, Drachsler concurrent search tree is a lock-based tree that acquires a large number of locks for each successful update operation that limits the performance. It has a better throughput on workloads where the lookup percentage is high and the main advantage of this implementation is that during deletion operation it can find the successor of node to be deleted in a single step ($\mathcal{O}(1)$) through successor pointer stored in the node structure. Finally, Bronson concurrent search tree outperforms Ellen and Drachsler concurrent search trees in most of our experiments (Figures 2.19 and 2.20) because of its balance. It is a relaxed balance AVL tree. However, it is also a lock-based complex algorithm which uses hand-over-hand locking technique and threads can block for a long time waiting an update operation to complete. As a result, its performance is very low in small trees with high contention (Figure 2.18).

As explained in [10] cache coherence is the most significant limiting factor for scalability for concurrent search algorithms on multiprocessor systems, since the number of cache line transfers increases with the number of threads. Thus, most concurrent search tree algorithms attempt to minimize the amount of cache traffic (cache line transfers) it performs during each operation which is directly associated with the number of stores on shared data structure. Stores provoke invalidation in cache lines according the cache coherence protocol and result to cache misses of future accesses. Generally, the fewer cache misses an algorithm generates, the better it scales.

Finally, the results explain that lock-based and lock-free algorithms are close in terms of performance, but in case of high number of threads (high contention) lock-free implementations provides better scalability than lock-based. Lock-free implementations also offer robustness. Moreover, the number of stores in a successful operation should be close to an asynchronous, sequential algorithm. The closer to the sequential algorithm that provides no synchronization, an implementation is, the higher performance it has.

Chapter 3

Transactional Memory

3.1 Transactional Memory (TM)

As multiprocessor systems became the dominant computing systems, the discovery of a non-blocking scheme for synchronization that would provide better scalability and would simplify the parallel programming was necessary. This need led to Transactional Memory (TM). An important benefit of transactional memory is that there are not locks and deadlocks.

Transaction memory attempts to simplify concurrent programming by allowing a group of load and store instructions to execute in an atomic way. Complex operations can be performed concurrently, in isolation from each other, with those operations either completing or being undone, as transaction, a model that developers are already familiar with from database programming. Transaction is a unit of work that either completes in its entirety or has no effect at all (is executed atomically).

Transactions must be serializable (appear to execute sequentially). Serializability is a kind of coarse-grained version of linearizability. Linearizability is a guarantee about single operations on single objects. Each method call of a given object should appear to take effect instantaneously between its invocation and response. For instance, once a write completes, all later reads should return the value of that writer or the value of a later writer. Once a read returns a particular value, all later reads should return that value or the value of a later write. Serializability, on the other hand, defines atomicity for entire transactions, that is, instruction groups in the code that include calls to one or more objects. It ensures that a transaction appears to take effect between the invocation of its first call and the response to its last call. Furthermore, it guarantees that the execution of a set of transactions over multiple objects is equivalent to some serial execution.

The idea of transactional memory is that during the execution of a transaction there is no need for synchronization. The underlying TM system can detect that a conflict has occurred because of a parallel execution of processes in multiple cores. A conflict occurs when two transactions perform conflicting operation to the same memory address. There are two types of conflicts:

1. A transaction writes to a memory address in which other processes perform a read or a write, too.
2. A transaction reads a memory address in which other processes perform a write.

If no conflicts detected during the execution of a transaction, then the underlying TM system attempts to persist the transaction's results and inform all other processors about the transaction's modifications (make all changes visible and permanent). This is a **transactional commit**. Otherwise, if conflicts are detected during the execution of a transaction, then the underlying TM system rolls back the current transaction, causes all the modifications made by the transaction to be discarded and revert the system to the previous stable state as if the transaction had never begun. This is a **transactional abort**. Therefore, we can conclude that the speed in which the conflicts are detected and the commits/aborts are executed is the most significant limiting factor for the performance of the underlying TM system.

TM system, depending on the implementation, can be divided into three categories, the Software Transactional Memory (STM), the Hardware Transactional Memory (HTM) and the Hybrid Transactional Memory.

3.1.1 Software Transactional Memory (STM)

Software transactional memory provides transactional memory semantics implemented exclusively in software, rather than as a hardware component. There is no use of hardware components in detecting conflicts or in performing transactional commits/aborts during transaction. It can be implemented as a lock-free algorithm or it can use locking and is a software runtime library. Some STM implementations released are: TiniSTM (C programming language), STMNet (C#), CL-STM (Common Lisp), STM Library (Haskell), Deuce, DSTM2 (Java), ScalaSTM (Scala).

The main advantage of software transactional memory is that it can be used in any platform/system, as there is no need for a particular hardware support. Furthermore, it is more flexible, as it permits implementation of a wider variety of more sophisticated algorithms and is easy to modify and evolve. However, transactional memory implemented entirely in software is slow and come with performance penalty, when compared to hardware solutions. Typically, detecting conflicts and performing the appropriate actions, after a transactional abort, to discard the modifications made by the transaction and revert the system to a previous stable state are very costly and time-consuming. As a result, software transactional memory can provide increased performance only in very particular cases.

3.1.2 Hardware Transactional Memory (HTM)

Hardware transactional memory was proposed as a performance improvement of software transactional memory. It consists of a full implementation of TM in hardware. Detecting conflicts and performing transactional commits or aborts are exclusively executed on hardware. The main purpose of HTM is to reduce the overhead of performing a trans-

action in a system. HTM can also have better power and energy profiles than STM and can provide strong isolation without requiring changes to non-transactional memory accesses.

As explained in [15], in attempt to implement HTM we have to add a transactional bit to each cache line's tag. Firstly, the transactional bit is unset, but when a value is placed in the cache line on behalf of a transaction, this bit is set and this entry is transactional. Modified transactional cache lines do not be written back to main memory before the transaction commits and invalidating a transactional cache line aborts the transaction. More specifically:

- If a transactional cache line is invalidated according to coherence protocol (e.g MESI), then the transaction is aborted. This invalidation indicated a synchronization conflict (another processor accessed on this cache line), either between two writes or a read and a write.
- If a modified transactional cache line is invalidated or evicted from the cache, the value is discarded (it must not be written to the main memory). While the transaction has not commit, we cannot evict tentative transactionally written values. In this case we must abort the transaction.
- If the cache evicts a transactional line, the cache coherence protocol cannot detect synchronization conflicts, since the cache line is no longer in cache. The transaction must be aborted, too.

Finally, when the transaction terminates and none of its transactional lines has been invalidated or evicted, the transaction commits, unsetting the transactional bits in its cache lines. If the transaction is aborted, its transactional cache lines are invalidated.

Although HTM is not so time-consuming as STM, it adds an important cost during transaction and especially in case of consecutive aborts. Nevertheless, the most limiting factor in HTM is the limited hardware resources. For instance, the size of the transaction is limited by the size of cache. Moreover, most operating systems flush the cache when a thread is descheduled, so the duration of the transaction may be limited by the time quantum of scheduling. As a result, HTM must be used for small transactions, whereas applications that need longer transactions should use STM or hybrid transactional memory. When a transaction aborts, the hardware should return a condition code indicating the reason of transactional abort. If the abort was due to a synchronization conflict (data conflict), the transaction should be retried. If the abort was due to a hardware resource exhaustion, there is no point in retrying the transaction. Another limitation in HTM is the usage of extensions in instructions of HTM. The code of the program must be rewritten for each processor that supports a different HTM implementation using different extensions each time. Finally, it is obvious that a program that employs an HTM implementation cannot be always executed in a platform which does not support HTM.

3.1.3 Hybrid Transactional Memory

This scheme is combination of both models, STM and HTM and provides benefits of both of them. For example, the papers [16], [17] propose an hybrid transactional memory, which implements TM in software so that it can use hardware TM (HTM) to boost

performance but it does not depend on HTM and does not expose programmers to any of its limitations. The papers [18] and [19] propose an approach, in which hardware is used to accelerate a TM implementation controlled fundamentally by software (hardware accelerated STMs).

3.2 Basic TM Characteristics

Different TM implementations combine different options of the basic TM characteristics. The basic characteristics for TM implementations are:

- **Data versioning**

  Transactional memory systems require a mechanism to manage the tentative writes in concurrent transactions. This mechanism is known as data versioning. Transactional writes require two copies of the written location to be stored: the committed (old) version of data (to be used by other processes/threads while the transaction in the current thread is executed, and if the transaction aborts) and the uncommitted (new) version of data (to be used when the transaction commits). There are two approaches of data versioning depending on the location in which the commited and the uncommitted version of data are stored:

  – *Eager versioning (undo-log based)*

    This approach is also known as direct update because it means that the transaction directly updates the data in memory. The transaction maintains an "undo log" holding the committed values that it has overwritten. This undo log works like a stack. Every time a memory location is modified, the committed (old) values are copied in the stack. Thus, when a conflict is arised during the transaction, the transaction aborts and the system rolls back with each step of the undo log being executed in reverse order to restore the previous original state of memory. Figure 3.1* shows an example of eager versioning.

  – *Lazy versioning (write-buffer based)*

    This approach is also known as deferred update because the updated are delayed until a transaction commits. The transaction maintains its tentative writes in a write-buffer in cache instead of directly writing to memory. When a transaction commits, it updates the actual memory locations from the copies of write-buffer. Since transaction's updates are maintained in the write-buffer, a read inside the same transaction must consult the write-buffer so that earlier writes are seen. If a transaction fails to commit due to a transactional conflict, the write-buffer is discarded and the transactions do not modify memory at all. Figure 3.2* depicts an example of lazy versioning.

* Image taken from http://15418.courses.cs.cmu.edu/spring2013/article/40.

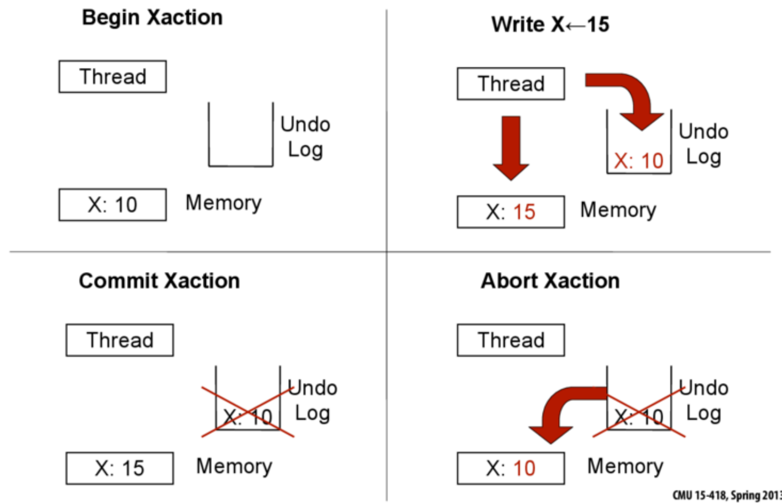


Figure 3.1: A simple example of eager versioning.

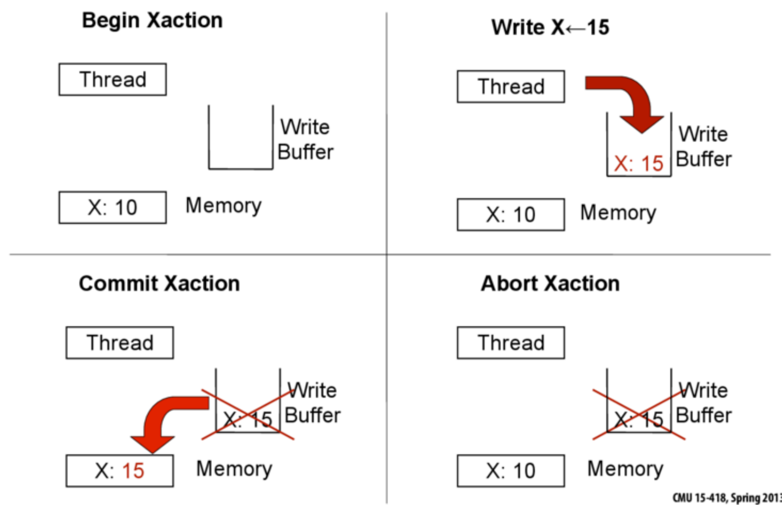


Figure 3.2: A simple example of lazy versioning.

Generally, commits are faster in eager versioning since new (modified) data are already stored in memory. On the other hand, rollback is faster (faster aborts) in lazy versioning as it just discards the write-buffer, while in eager versioning the committed (old) data has to be copied from the undo log to memory. Furthermore, in lazy versioning each store requires only one write to buffer, whereas in eager versioning it requires a write to memory as well as to the undo log. Finally, in case of a crash during transaction, memory will be in an inconsistent state in eager versioning, in contrast to lazy versioning that handles faults in a better way, since the memory is in a consistent state during transaction.

- **Conflict detection**

  Conflicts during transactions must be detected and handled properly to ensure correctness. There are two types of conflicts: read-write conflict and write-write conflict. A read-write conflict appears when a transaction reads an address, which was written to by another pending transaction. Similarly, a write-write conflict appears when two (or more) pending transactions write to the same address in memory. To achieve conflict detection the system keeps track of each transaction's read set and write set, which are the addresses read from or written to in each transaction. There are two policies for conflict detection:

  – *Pessimistic detection*

    This policy is also known as eager conflict detection. It attempts to detect conflicts early, as soon as a load or a store is requested. If a conflict is detected, then the contention manager (contention manager is responsible for making transactions look as if they are sequentially executed) either aborts the transaction, or stalls one of the transactions until the other completes. There are various priority policies to determine which transaction gets priority and handle common case fast. Figure 3.3[*] shows some pessimistic detection examples.

  – *Optimistic detection*

    This policy is also known as lazy conflict detection. It attempts to check conflicts only at commit time. Before committing, the write-set is communicated to all other pending transactions in order to check conflicts. Usually, on a conflict, the committing transaction has priority and other transactions may abort later on. Figure 3.4[*] depicts some optimistic detection examples.

  Finally, there are hybrid policies that use optimistic and pessimistic schemes together. For example, several STM systems use optimistic policy for reads and pessimistic for writes. Comparing the two policies, in pessimistic conflict detection, there is no forward progress guarantee and it may lead to a livelock (Figure 3.3 Case 4).

[*] Image taken from http://15418.courses.cs.cmu.edu/spring2013/article/40.

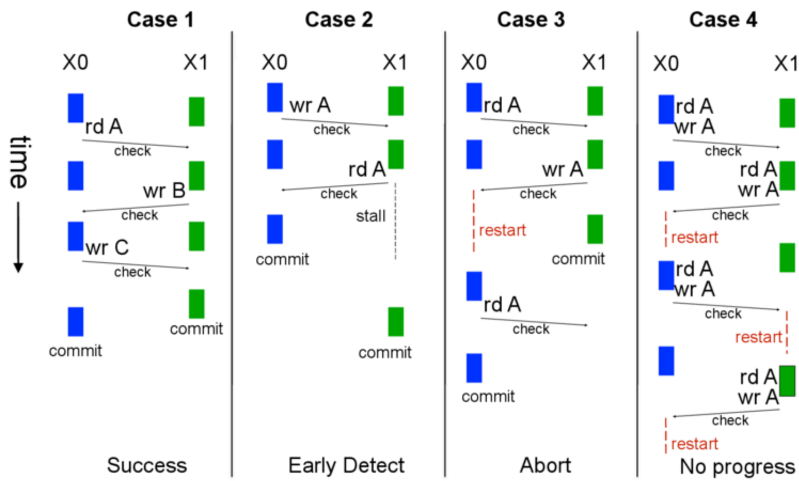


Figure 3.3: A discussion of pessimistic detection.

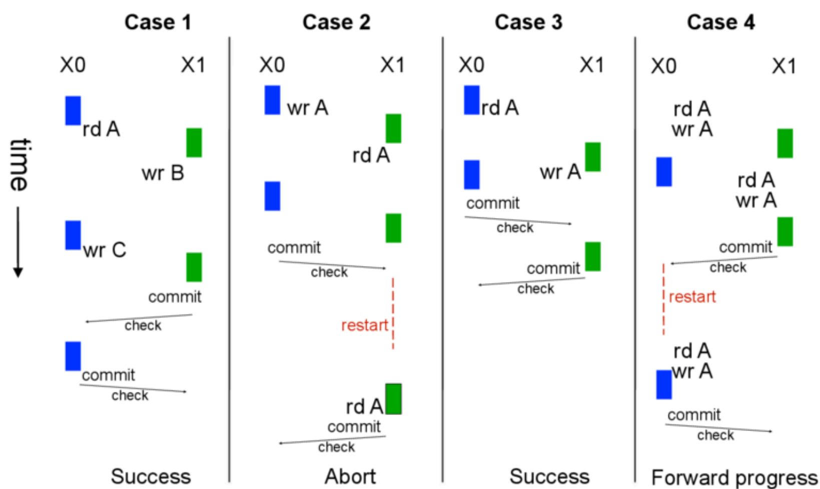


Figure 3.4: A discussion of optimistic detection.

- **Conflict resolution**

  The conflict is resolved when the underlying system take some action to avoid conflicts (e.g stall or abort one of the conflicting transactions). Eager conflict detection must resolve the conflict as soon as the transaction requests a load or a store that conflicts with one or more other pending transactions. The resolution policy can stall the transaction, abort the transaction, or abort others. Lazy conflict detection must resolve the conflict as soon as a transaction, that conflicts with one or more transactions, attempts to commit. The resolution policy can abort all others, stall or abort the committing transaction.

- **Isolation**

  As defined in [20] isolation requires that execution of a transaction does not affect the result of concurrently executing transactions. *Strong isolation* implies that transactional blocks are isolated from other transactional blocks and from concurrent non-transactional accesses. This means that a conflict is detected even if the conflicting access occurs in a non-transactional code. On the other hand, *weak isolation* implies that transactions are isolated only from other transactions. Therefore, in a system with weak isolation a non-transactional read may see the state of an incomplete transaction and a non-transactional write may appear to occur in the middle of a transaction.

- **Granularity**

  Transaction granularity is the unit of storage over which transactional memory system detects conflicts. There are three alternatives for transaction granularity, object granularity, word granularity and cache line granularity. *Object granularity* detects a conflicting access to an object even if the transactions referenced different fields. *Word granularity* is also known as *block granularity* and detects conflicting accesses to a memory word or adjacent (fixed-size group of words). And *cache line granularity* detects conflicting accesses to a cache line even if transactions modify disjoint parts of a cache line. Most HTM systems detect conflicts at cache line granularity, while most STM systems operate on an object granularity.

- **Best effort**

  This characteristic appears only in real HTM implementations. Using only transactional mode, no forward progress is guaranteed. A transaction may always fail to commit (Figure 3.3 Case 4) and therefore a non-transactional fallback path is necessary.

- **Conflicts**

  A transaction may fail to commit (abort) because of different reasons of conflicts. In STM systems a transaction aborts due to data conflicts, while in HTM systems a transaction may abort for several reasons like data conflicts, capacity aborts, explicit aborts and interrupts.

- *Data conflict*: This conflict appears when two or more threads perform conflicting operations to the same data. For example, when another process/thread writes to a memory location that has been added to the transaction's read or write set.
- *Capacity abort*: Transactional buffers of TM system have a fixed size for each process/thread. Thus, transaction's read and write set have limited capacity. When a transaction exceeds the maximum (write or read) buffering capacity imposed by the TM implementation, the transaction fails to commit due to a capacity abort.
- *Explicit abort*: This type of abort occurs when the programmer explicitly aborts the transaction. For instance, in real HTM implementations that use best effort, the code in the fallback path includes acquiring a global lock to protect the critical section. Thus, an explicit abort is performed at the beginning of a transaction if this lock is checked and found to be taken.
- *Other*: A transaction may abort due to several other reasons including interrupts, unsupported instructions, system calls e.t.c.

3.2.1 Real TM implementations

The following examples constitute real TM implementations.
HTM implementations

- Lazy + optimistic: Stanford TCC
- Lazy + pessimistic: MIT LTM, Intel VTM, Sun's Rock
- Eager + pessimistic: Wisconsin LogTM

STM implementations

- Lazy + optimistic (rd/wr): Sun TL2
- Lazy + optimistic (rd)/pessimistic (wr): MS OSTM
- Eager + optimistic (rd)/pessimistic (wr): Intel STM
- Eager + pessimistic (rd/wr): Intel STM

3.3 Intel's Haswell HTM

As already mentioned, there are different TM implementations which combines the above characteristics and manage differently the transactions. For example, Sun's Rock combines lazy versioning with pessimistic conflict detection and Stanford TCC uses lazy versioning and optimistic conflict detection. In this section we will further analyze an HTM implementation that Intel provides in Haswell processors. Intel announced that its Haswell architecture would include hardware support for transactional memory in 2013 and Haswell became the first x86 processor to feature hardware transactional memory.

Haswell's HTM implementation uses lazy data versioning, pessimistic (eager) conflict detection, operates on cache line granularity (64 bytes), provides strong isolation and is a best-effort implementation.

The programmer have to mark the block of code which have to be executed as a transaction. Thus, the ISA (Instruction Set Architecture) of the processor has been extended with a set of instructions that allows programmer to use the HTM infrastructure. While this block of code is executed as a transaction, the system is in transactional mode. In transactional mode the memory addresses that the transactional code accesses, are transferred in cache memory and are separated in two sets, the read and the write set (consist of all memory addresses in which the transaction reads and writes, respectively). Each cache line in L1D and L2 cache contains bits that indicate whether the line belongs to the read set or the write set. Any transactional data must stay in L1D cache (or L2) and not be evicted to the L3 or memory, until the transaction commits.

While the transaction is executed, possible conflicts, that may arise from the parallel execution of different processes/threads, can be detected. This conflicts can be detected using cache coherence protocol (e.g MESIF). Briefly, when a process/thread that runs in a processor, reads or writes to a memory address, cache coherence protocol is responsible to communicate with other processors that have already stored in their L1D cache this memory address and inform them for the modification. Furthermore, since the cache coherence protocol informs immediately other processors for modifications, there is no need for additional communication among processes/threads that run concurrently, something that would limit the memory bus bandwidth.

If during transaction a conflict is detected, the transaction aborts and the HTM implementation have to roll back the current transaction, discard the modifications made by the transaction and revert the system to the previous initial state as if the transaction had never begun. This is simple for Haswell's HTM implementation, as all modifications made by the transaction are stored in L1D cache (and/or L2) and the main memory of the system has not been updated. As a result, the underlying TM system invalidates all the transactional cache lines of L1D (and/or L2) cache and main memory remains immutable. However, if a transaction attempts to commit and no conflict has been detected, the underlying system has to transfer the modified transactional cache lines from cache to main memory.

Finally, there is an important limitation in Haswell's HTM implementation. The cache detects concurrent accesses at a cache line granularity. For instance, the case of two threads that write to adjacent elements of a large array will often cause the transaction to fail (abort), because the hardware cannot distinguish between two accesses to the same address, and two accesses to different addresses in the same cache line.

3.3.1 Transactional Synchronizations Extensions (TSX)

In order to implement HTM in Haswell processors, its instruction set architecture (x86) has been extended with a set of instructions to ease the development and improve the performance of existing programming models. In this system it is more convenient the use of C or C++ programming language. Haswell's transactional support, which Intel is call-

ing Transactional Synchronization Extensions (TSX) provides two software interfaces. The first, called Hardware Lock Elision (HLE) allows easy conversion of lock-based programs into transactional programs in a way that is compatible with current processors. The second, called Restricted Transactional Memory (RTM) is a more complete transactional memory implementation that allows programmers to define transactional regions in a more flexible manner than is possible with HLE. RTM also requires programmer to provide an alternate code path (fallback path) in case that transactional execution is not successful, since the hardware provides no guarantees as to whether an RTM region will ever successfully commit transactionally.

These extensions can help achieve the performance of fine-grained locking synchronization through a coarse-grained locking implementation in the code. Moreover, these extensions allow locks around critical sections and perform serialization of parallel executions of critical sections only when this is necessary. Multiple processes/threads that execute critical sections and do not perform any conflicting operations can proceed simultaneously without serialization. Although the software uses a global lock to protect critical sections, the hardware is allowed to recognize that processes/threads do not interfere with one another.

The main difference between HLE and RTM is that software written using HLE can run both on legacy hardware without TSX (in this case the critical section is executed directly in lock mode) and new hardware with TSX, while software written in RTM cannot run in a processor that does not support TSX. However, RTM offers more flexibility concerning the actions that can be done after a transactional abort. The programmer defines a memory address that points out the code that will be executed in case of abort (fallback handler). To employ TSX the programmer must have a complier gcc-4.8x (or a later version) and include in his program the library "immintrin.h". Otherwise, in case of an older version of gcc, the programmer must include the library "rtm.h".

Hardware Lock Elision (HLE)

Hardware Lock Elision is a simple way of deploying transactional memory in existing code. The idea of HLE is to remove locks and let CPU worry about consistency. Instead of assuming that a process/thread always protect the shared data from other threads, it can be assumed that the other processes/threads will not overwrite the variables that the current process/thread is working on (in the critical section). If another process/thread overwrites one of those shared variables, the whole process will be aborted by the CPU, and the transaction will be re-executed with a traditional lock.

HLE introduces two new instruction prefixes, named XACQUIRE and XRELEASE that are used to denote the bounds of critical section. XACQUIRE is a prefix for instructions that acquire a lock and it indicates the start of critical section (region for lock elision). When a process/thread acquires a lock with an XACQUIRE instruction, the lock is not actually acquired. The write operation is ignored, but the memory address of the lock instruction is added to the read set of the transaction, so the transaction will fail if something else writes to that address. The process/thread then enters transactional ex-

ecution and continues on to the instructions inside the critical section, adding memory addresses to transaction's read and write set. The current process/thread will still think it has obtained the lock, but many processes/threads will be allowed to run simultaneously and make non-conflicting accesses to shared data.

Execution continues until the XRELEASE instruction. XRELEASE is a prefix that is used for the instruction that releases the lock address, and it marks the end of the critical section. When the processor reaches XRELEASE instruction, it attempts to commit the transaction. If it succeeds, the critical section was executed without acquiring or releasing the lock (none of the memory operations conflicted). If the transaction fails (when a conflict occurs), the processor will restore the architectural register state prior to XACQUIRE and discard any writes from the critical section. The process/thread will execute the critical section again, with the standard pessimistic locking behavior. In this case, it actually acquires the global lock. So the programmer can use coarse-grained locking as a "fall back" solution, and HLE can exhibit as much parallelism as is present in the access patterns, not in the locking designs.

As already mentioned, the software written using HLE can also run to hardware without TSX. The system is backwards compatible. The programmer can use the new TSX enabled library and gets the benefits of TSX if his program is executed on Haswell or a later Intel CPU. Every other processor will ignore the prefix and just operate on the lock, the traditional lock-based behavior. The prefixes of the instructions XACQUIRE and XRELEASE are treated as nops.

The listing 3.1 presents an example of elision of a TAS lock:

Listing 3.1: Example: elision of a TAS lock

```
1  /* Traditional lock implementation */
2  /* acquire lock */
3  while(__sync_lock_test_and_set(&lock_var) == 0)
4   /* do nothing */;
5  ... Critical section with lock acquired ...
6  /* release lock */
7  __sync_lock_release(&lock_var);
8
9
10 /* HLE implementation */
11 /* elide lock */
12 while(__hle_acquire_test_and_set(&lock_var) == 0)
13 /* do nothing */;
14 ... Critical section with lock acquired ...
15 /* release lock */
16 __hle_release_clear(&lock_var);
```

Restricted Transactional Memory (RTM)

Restricted Transactional Memory (RTM) is an alternative implementation to HLE which gives the programmer the flexibility to specify a fallback code path that is executed when a transaction cannot be successfully executed. There are four new instructions, XBEGIN, XEND, XTEST and XABORT. The programmer marks the block of code that he wants to be executed atomically (critical section) using the instructions XBEGIN and XEND. XBEGIN and XEND mark the start and the end of the critical section, respectively. When the process/thread reaches the XEND instruction in the code, the transaction commits and the memory is updated according to the modifications made by the transaction. The instruction XTEST returns 1 if the process/thread is in transactional mode, otherwise it returns 0 and with XABORT(status) instruction the programmer can explicitly abort the transaction (as if a commit have been unsuccessful). The status is used to indicate the reason for transactional abort. An explicit abort instruction is useful when the programmer can determine that a transaction is going to fail, without any help from the hardware. Aborting the transaction early can also help reduce the power penalty.

If a conflict occurs during transaction, it may trigger an abort. After a transactional abort the fallback handler is responsible for the instruction that the process/thread will execute to resume the execution. The programmer defines the memory address of the code to be executed in case of transactional abort (fallback address). The fallback address is exactly the next instruction after XBEGIN. XBEGIN returns a value that indicates if the process/thread is in transactional mode or if the transaction has been aborted. As a result, the EAX register is updated according to the transaction's status (Figure 3.5). The programmer can check if the transaction has started or if it has been aborted performing a logical calculation and between the return value of XBEGIN and the following constants:

- _XBEGIN_STARTED: Transaction has successfully begun.
- _XABORT_CONFLICT: Transaction abort due to a memory conflict with another thread.
- _XABORT_CAPACITY: Transaction abort due to the transaction using too much memory.
- _XABORT_EXPLICIT: Transaction was explicitly aborted with _xabort. The parameter passed to _xabort is available with _XABORT_CODE(status).
- _XABORT_RETRY: Transaction retry is possible.
- _XABORT_DEBUG: Transaction abort due to a debug trap.
- _XABORT_NESTED: Transaction abort in an inner nested transaction.

Some causes of abort may always result to transactional abort. As a result, each time we execute the critical section in transactional mode using the HTM implementation, the transaction always fails and there is no progress in our program. For example, if transaction's read and write set exceed the size of cache, the transaction will always abort because of capacity aborts. In this case the programmer should use a back-off mechanism like a fallback path. The fallback path is an alternative implementation in the code of the program that does not employ RTM and is most likely a piece of code that does coarse-grained

EAX register bit position	Meaning
0	Set if abort caused by XABORT instruction.
1	If set, the transaction may succeed on a retry. This bit is always clear if bit 0 is set.
2	Set if another logical processor conflicted with a memory address that was part of the transaction that aborted.
3	Set if an internal buffer overflowed.
4	Set if debug breakpoint was hit.
5	Set if an abort occurred during execution of a nested transaction.
23:6	Reserved.
31:24	XABORT argument (only valid if bit 0 set, otherwise reserved).

Figure 3.5: Transaction's status is captured to EAX register's bits.

locking. After a specified number of aborts in a transaction the programmer can choose the execution of the fallback path instead of transaction. This implementation is necessary to guarantee progress in program's execution.

The fallback path is usually implemented as coarse-grained locking code that uses a global lock. In this case this global lock has to be added to transaction's read set. If the global lock does not be added to transaction's read set, it results to coherence problems. Figure 3.6 presents an example of two threads that attempt to execute the same code concurrently. The first thread enters the critical section acquiring the global lock and the second thread using RTM implementation. In this example the second thread will not detect any conflict and its transaction will commit updating the value of the "count" variable, while the first thread will not be informed about this modification. This constitutes a coherence problem. As a consequence, the programmer is responsible to add the global lock to transaction's read set. To achieve this the value of the global lock must be read without locking the global lock. If at the beginning of a transaction the global lock is used from another thread, the programmer must explicitly abort the transaction (explicit abort). Listing 3.2 is an example of adding a pthread_spinlock in transaction's read set, since the code reads its value, and in case that this is not free the transaction explicitly aborts via XABORT instruction.

Listing 3.2: Example: adding global lock to transaction's read set

```
1  if ((int)spin_lock != 1)
2              _xabort();
3  if (pthread_mutex_t.__data.__lock != 0)
4              _xabort ();
```

Finally, Listing 3.3 describes an RTM example in C programming language.

Listing 3.3: An RTM example

```
1  int aborts = MAX_TX_RETRIES;
2  lock_t = fallback_global_lock;
```

```
3  start_tx:
4          int status = TX_BEGIN();
5          if(status == TX_BEGIN_STARTED){
6                  if (fallback_global_lock is locked)
7                          TX_ABORT();
8          ... Critical Section ...
9          TX_END();
10         }else{ /* status != TX_BEGIN_STARTED */
11                 if (--aborts > 0)
12                         /* retry transaction */
13                         goto start_tx;
14                 acquire_lock(fallback_global_lock);
15                 ... Critical Section ...
16                 release_lock(fallback_global_lock);
17         }
```

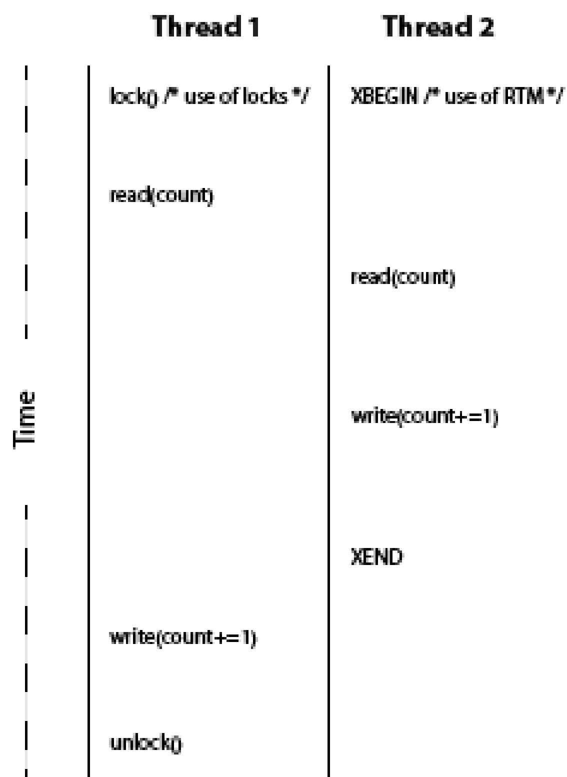


Figure 3.6: A parallel execution of two threads using RTM that results to coherence problems.

Chapter 4

A parallelization of Dijkstra's algorithm

4.1 Dijkstra's algorithm

4.1.1 Algorithm's history

Dijkstra's algorithm (also known as shortest path algorithm) is a fast algorithm for finding the shortest paths between nodes in a graph, which may represent a road network. It was conceived by a Dutch computer scientist from Netherlands Edsger Wybe Dijkstra (May 11, 1930 – August 6, 2002) in 1956 and published three years later [21]. Edsger Wybe Dijkstra is also known for his many essays on programming and received the A. M. Turing Award (widely considered the most prestigious award in computer science) in 1972.

As the history of shortest paths algorithms shown in figure 4.1 discloses, Dijkstra's algorithm is a simpler and faster version of Ford's algorithm. Wikipedia describes that Dijkstra thought about this algorithm when working at the Mathematical Center in Amsterdam in 1956 as a programmer to demonstrate capabilities of a new computer called ARMAC. His main purpose was to present both a problem as well as an answer, that would be produced by computer, that people could understand. He designed the shortest path algorithm and implemented it for ARMAC computer for a slightly simplified transportation map of 64 cities in Netherlands. A year later, he came across another problem from hardware engineers working on the institute's next computer: minimize the amount of wire needed to connect the pins on the back panel of the machine. As a solution, he re-discovered the algorithm known as Prim's minimal spanning tree algorithm [22] and published his algorithm in 1959, two years after Prim.

The algorithm exists of many variants. Dijkstra's original algorithm finds the shortest path between two nodes, but the most popular variant of this algorithm fixes a single node as the "source" node and finds the shortest paths from the source to all other nodes in the graph, producing a shortest-path tree. Furthermore, in some fields (artificial intelligence)

Dijkstra's algorithm or a variant of it is known as uniform-cost search and formulated as an instance of the more general idea of best-first search.

Shimbel (1955)	Information networks.
Ford (1956).	RAND, economics of transportation.
Leyzorek, Gray, Johnson, Ladew, Meaker, Petry, Seitz (1957).	Combat Development Dept. of the Army Electronic Proving Ground.
Dantzig (1958).	Simplex method for linear programming.
Bellman (1958).	Dynamic programming.
Moore (1959).	Routing long-distance telephone calls for Bell Labs.
Dijkstra (1959).	Simpler and faster version of Ford's algorithm.

Figure 4.1: Early history of shortest paths algorithms.

4.1.2 Description

Dijkstra's algorithm is a greedy algorithm that solves the single-source path problem when all edges have non-negative weights. This is asymptotically the fastest known single-source shortest-path algorithm for graphs with unbounded non-negative weights. The algorithm is based on the observation that any subpath of any shortest path is itself a shortest path (optimal substructure). Extending this idea we observe the existence of a shortest path tree in which the distance from source to vertex v is the length of shortest path from source to vertex in original tree. The length of a path $p = (v0, v1, ..., vk)$ is the sum of the weights of its constituent edges: $length = \sum_{i=1}^{k} w(v_{i-1}, v_i)$, where the function $w : E \to \mathbb{R}$ maps edges to the real-valued weights.

Intuitively, the algorithm reports the vertices in increasing order of their distance from the source vertex. Exploring a new vertex means exploring the vertex that has the smallest distance. This is why the algorithm uses the distance from the source to the vertex as the priority. Secondly, the algorithm constructs the shortest path tree edge by edge. At each step adding one new edge corresponds to the construction of shortest path to the current new vertex. The new edge is added to the shortest path to the current new vertex, if the new path from the source to the vertex is shorter than the previous distance from the source to vertex. The process by which an estimate of the distance from source to vertex is updated is called **relaxation**.

For a graph, $G = (V, E)$ where V is a set of vertices and E is a set of edges. Dijkstra's algorithm keeps two sets of vertices: S the set of vertices whose shortest paths from the source have already been determined and $V \backslash S$ the remaining vertices (unvisited set). The algorithm in steps is:

1. Set S to empty.
2. Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE (∞) and the distance value as 0 for the source vertex.

3. While there are still vertices in $V \backslash S$ (unvisited)
   (a) Choose an unvisited vertex u from $V \backslash S$ that has the minimum distance value from the source.
   (b) Include vertex u to the set S.
   (c) Relax all adjacent vertices of u that are still in $V \backslash S$. To achieve this, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge $u - v$, is less than the distance value of v, then update the distance value of v.

The listing 4.1 is a pseudocode for Dijkstra's algorithm using priority queue. The Q set is the set of unvisited vertices ($V \backslash S$) that is implemented as a priority queue and the arrays dist, prev have the distance from the source to a vertex v and the previous node of vertex v in the optimal path from source, respectively. The function add_with_priority() adds an element to the queue with an associated priority (minimum distance from source), the function extract_min() removes the element from the queue that has the highest priority (more specifically the vertex with the minimum distance from source), and return it, and the function decrease_priority() updates the priority of an element in the queue.

Listing 4.1: Dijkstra's algorithm

```
1   function dijkstra(graph, source):
2           dist[source] ← 0 //Initialization
3
4           create vertex set Q //Set of unvisited vertices
5
6           for each vertex v in Graph:
7                   if v ≠ source
8                   dist[v] ← INFINITY //Unknown distance from source to v
9           prev[v] ← UNDEFINED //Predecessor of v
10
11                  Q.add_with_priority(v, dist[v])
12
13
14          while Q is not empty: //The main loop
15                  u ← Q.extract_min() //Remove and return best vertex
16                  for each neighbor v of u: //Only if v that is still in Q
17                          sum = dist[u] + length(u, v)
18                          if sum < dist[v] //A shorter path has been found
19                                  dist[v] ← sum
20                                  prev[v] ← u
21                                  Q.decrease_priority(v, sum)
22
23          return dist[], prev[]
```

4.1.3 Complexity

The simplest implementation of the algorithm stores the vertex set as an ordinary linked list or array. Extract_min() takes $\mathcal{O}(V)$ time and there are |V| such operations. Therefore, a total time for extract_min() in while loop is $\mathcal{O}(V^2)$. Since the total number of edges in all the adjacency list is |E|, the for loop iterates |E| times with each iteration

taking $\mathcal{O}(1)$ time. Hence, the complexity of the algorithm with an ordinary linked list or array implementation is $\mathcal{O}(V^2 + E) = \mathcal{O}(V^2)$.

For sparse graphs (graphs with fewer edges), the algorithm can have a better complexity by storing the vertex set in the form of adjacency lists and using a self-balancing binary search tree, binary heap, pairing heap, or Fibonacci heap as a priority queue. This results to a more efficient implementation of extract_min() function. Extract_min() takes $\mathcal{O}(\log V)$ time and there are |V| such operations. The function decrease_priority() (or decrease_key()) takes $\mathcal{O}(\log V)$ in case of a self-balancing binary search tree or a binary heap and $\mathcal{O}(1)$ in case of a fibonacci heap for each of the |E| edges. Thus, the running time of the algorithm with self-balancing binary search tree or binary heap provided is $\mathcal{O}((E + V)\log V)$ and with fibonacci heap provided is $\mathcal{O}(E + V\log V)$.

4.1.4 Proof of correctness

Lemma: When a vertex u is added to S set (visited nodes), then $dist[u] = [s, u]$, where $[s, u]$ the length of the shortest path from the source to the vertex u.

Proof: Suppose that the algorithm first attempts to add a vertex u to the set S for which $dist[u] \neq [s, u]$. Then, $dist[u] > [s, u]$.

Consider the shortest path (Figure 4.2) from source s to vertex u ($s \epsilon S$ and $u \epsilon V \backslash S$). Let (x,y) be the edge taken by the path, where $x \epsilon S$ and $y \epsilon V \backslash S$ (it may be that $x = s$ and/or $y = u$).

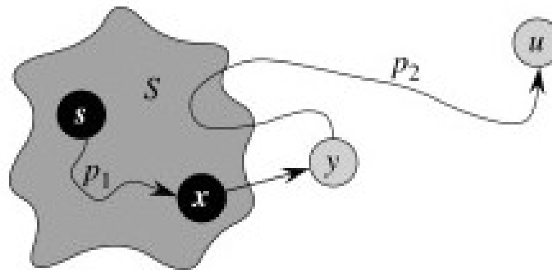


Figure 4.2: Proof of corectness of Dijkstra's algorithm. When a vertex u is added to S set (visited nodes), then $dist[u] = [s, u]$.

Having done the relaxation in vertex x we can conclude that

$$dist[y] \leq dist[x] + w[x, y], \tag{4.1}$$

where the function $w : E \rightarrow \mathbb{R}$ maps edges to the real-valued weights.

By hypothesis x is in the set S, so:

$$dist[x] = [s, x]. \tag{4.2}$$

Since <s,...,x,y> is a subpath of a shortest path, by 4.2

$$(s, y) = (s, x) + w(x, y) = dist[x] + w(x, y). \tag{4.3}$$

By 4.1, 4.3

$$dist[y] <= (s, y).$$

Therefore,

$$dist[y] = (s, y).$$

So $y \neq u$, as we suppose that $dist[u] > (s, u)$.

As a result,An example of a graph.

$$dist[y] = (s, y) < (s, u) <= dist[u].$$

Thus, y would have been added to S set before u, since it has a smaller estimate of the distance from the source. This contradicts with the assumption that u is the next vertex to be added to the set S.

By the lemma, $dist[u] = (s, u)$ when is added to the set S and at the end of the algorithm, all vertices are in the set S and all distance estimates are optimal.

4.1.5 Applications

As described above, the algorithm finds the shortest path between a node and every other. It can be also used for finding the shortest path from a single node to a single destination by stopping the algorithm when the shortest path to the destination node has been determined. For instance, in a road network, supposing that the cities are represented as the nodes of the graph and driving distances between pairs of cities connected by a direct road are represents as edges paths, Dijkstra's algorithm can find the shortest route between one city and all other cities. Thus, the shortest path algorithm is a widely useful problem-solving model used in network routing protocols, VLSI design, social networks and TeX typsetting. Figure 4.3 presents some applications of Dijkstra's algorithm.

4.2 The concept of Helper Threads

Helper threads is an optimization technique used in non traditional parallelism to accelerate a program and provide performance speedups. Helper threads are "assist" threads that perform certain critical computations on behalf of a main thread in order to help the main ("master") thread and reduce its tasks. Typically, this optimization has been exploited either to prefetch future data accesses or to precompute the outcome of blocks of code that would otherwise be executed by the main thread.

To improve the performance of an application program using the concept of helper threads, there are several key issues that need to be taken into consideration. First, in

Maps
Robot navigation
Texture mapping
Typesetting in TeX (e.g LaTeX)
Urban traffic planning
Optimal pipelining of VLSI chip
Subroutine in advanced algorithms
Telemarketer operator scheduling
Routing of telecommunications messages
Approximating piecewise linear functions
Network routing protocols (OSPF, BGP, RIP)
Exploiting arbitrage opportunities in currency exchange
Optimal truck routing through given traffic congestion pattern

Figure 4.3: Applications of Dijkstra's algorithm.

hyper-threaded processors some structures are shared or partitioned in between logical processors in multi-threading mode, and thus, resource contention can be an issue. As a consequence, helper threads can be invoked judiciously to avoid potential performance degradation due to the increased resource contention. Second, the program behavior changes dynamically, and hence helper thread invocation should be adaptable. For instance, a particular load might experience a significant number of cache misses over the total program execution, but the temporal distribution of those misses might not be uniform. As a result, a helper thread should be able to detect the dynamic program behavior at runtime. Finally, to adapt to the dynamic behavior, helper threads need to be activated and synchronized frequently. Thus, a low overhead thread synchronization mechanism is needed. Compared to traditional multi-threading technique where each tread should be executed in a pre-defined order to guarantee the correctness of the program, helper threads only affect the performance speedup of the program. Accordingly, a helper threads can be deactivated whenever it does not improve the performance of the main thread. Finally, dynamic program behaviors can be effectively captured at runtime and various dynamic optimizations can be applied.

4.3 Parallelizing Dijkstra's algorithm

4.3.1 Introduction

This section describes a parallelization of Dijkstra's algorithm presented in the papers [23], [24]. As explained in these papers, dijkstra's algorithm is based on the iterative extraction of nodes (vertices) from a priority queue. This property limits the explicit par-

allelism of the algorithm and any attempt to utilize the remaining parallelism results to performance degradation due to synchronization overheads. Thus, the two major issues inherent to the algorithm is the limited explicit parallelism and excessive synchronization. To deal with this problems the authors of the papers employed the concept of Helper Threads (HT) to extract more parallelism and Transactional Memory (TM) as a means of concurrent accesses to shared data structures.

The authors of [23], [24] chose the idea of Helper Threads to coarsen the granularity of parallelism. The key idea is that helper threads will offload operations from the main thread. More specifically, the main thread performs many relaxations of the nodes of the priority queue. Therefore, parallel helper threads can simultaneously relax the distances of several nodes. While the main thread extracts and updates the neighbors of the head of the priority queue, k helper threads can update the neighbors of the next k nodes in the priority queue in order to offload operations of the main thread in its next iteration.

Finally, TM system is a promising approach for dynamic data structures and applications with independent threads providing performance gains. The programmer is able to envelop blocks of code within a transaction, indicating that within this segment of the code exist accesses to memory addresses that may be performed by other threads as well. The TM system monitors the concurrent transactions of the threads and if two or more perform conflicting accesses, it resolves the conflict. In the case of non-conflicting accesses, TM systems perform the appropriate accesses with no overhead.

4.3.2 The algorithm

The algorithm exploits the basic property of Dijkstra's algorithm: the relaxations result to monotonically decreasing values for the distances of unvisited nodes until each distance reaches its final optimal (minimum) value. When a node is inserted in the queued set (its distance from the source is no longer infinite) its neighbors could also be relaxed to newer updated values. The original algorithm does not take into consideration this property and avoids computing intermediate distances that will be overwritten by updating only the neighbors of the extracted node. The idea is that Helper Threads can relax neighbors belonging to the queued set. Some of these relaxations will be offloaded by the main thread.

Helper threads perform relaxations to the top k positions in the queue and the corresponding nodes might have already obtained their optimal distance from source with some probability. Therefore, when helper threads read their distances and relax their outgoing edges, the corresponding neighbors related with these outgoing edges may obtain their optimal distance from source, as well. As a result, when in the next iteration the main thread checks these nodes (vertices), it will not perform any relaxations. On the other hand, a helper thread may perform a relaxation to a node that has not obtained its optimal (minimum) distance yet. In this case the node will eventually be set to its optimal minimum distance, when it will be examined by the main thread later on.

The main thread operates like in the sequential version. In each iteration it extracts the minimum vertex from the priority queue and performs its relaxations. At the same

time, the k-th helper thread reads the tentative distance of the k-th vertex in the queue and attempts to relax its outgoing edges according to this value. When the main thread finishes all its relaxations, it notifies helper threads to stop their relaxations, and they all proceed to the next iteration. This scheme is demonstrated in Figure 4.4 *.

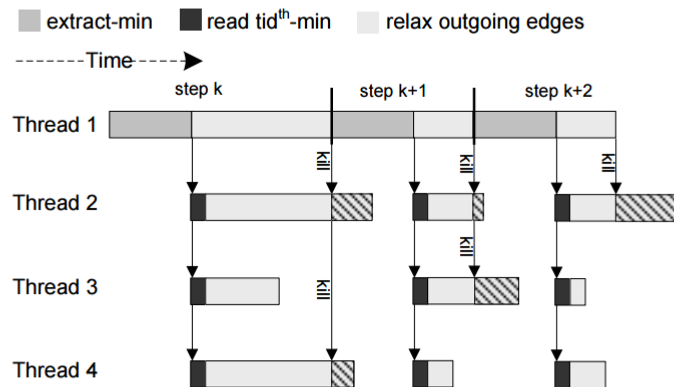


Figure 4.4: Execution pattern of the HT scheme.

In case that helper threads are forced by the main thread to stop their computations and proceed with it to the next iteration, it is possible that at this time a helper thread might have updated only some of the neighbors of its vertex, leaving the rest neighbors with their old distances. Nevertheless, this is not a problem since all neighbors of this vertex will eventually obtain their optimal distances when the vertex reaches at the top of the priority queue.

The code executed by the main and helper threads is shown in listings 4.2 and 4.3, respectively. In each iteration, the main thread extracts the vertex with the high priority (minimum distance from source) from the priority queue. At the same time, helper threads wait (spinning in a while loop) until the main thread finishes its extraction. Subsequently, each helper thread reads (without extracting) one of the top k vertices in the queue. This is done by ReadMin() function. In the next step, all threads, both the main and helper threads, perform the appropriate relaxations related with the outgoing edges of the vertices they have taken over. As explained above, helper threads offload relaxations of the main thread and thus, it will evaluate the expression of line 7 in Listing 4.2 as true fewer times and will not need to perform the operations of lines 8-10.

The proposed scheme should provides atomicity because a conflict can arise when two or more threads update concurrently the same neighbor, or update different neighbors but change the same part of the queue. To achieve atomicity updates to the queue via the DecreaseKey() function, as well as updates to the shared distance and predecessor arrays (d[], p[] respectively) are enclosed within a single transaction for both main and helper

* Image taken from [24].

threads. In this way, when a conflict arises, only one thread will be allowed to proceed, commit the transaction and perform the update to the queue, while the rest will have to repeat their work.

As already mentioned, when the main thread finishes its relaxations, it notifies helper threads to stop and proceed all to the next iteration. To implement this, the algorithm employs transactional memory (TM). More specifically, when the main thread completes the iteration of the inner loop for relaxations (line 4), it sets the notification variable "done" to 1. This means that the main thread will proceed to the next iteration for the next vertex and it also forces all helper threads to stop and follow, terminating their computations that they were performing on the queue. Since helper threads are in transactional mode and "done" variable is in their read sets, they will abort and they will retry the transaction. However, when helper threads will attempt to perform a new transaction for their work, they will find, with some strong probability, the "done" variable set to 1 and then they will stop their relaxations for the remaining neighbors in the inner loop and will proceed to the next iteration of the outer loop. If the main thread performs the ExtractMin() function too quickly and "done" variable will set back to 0, helper threads will miss the last notification, continuing from the point where they have stopped. This does not affect the guarantee of correctness. Although helper threads may update the distances of the neighbors with a suboptimal value, these will be overwritten with the optimal value when the vertices examined by helper threads reach at the top of the priority queue.

Finally, the main purpose of the algorithm is to employ helper threads only to offload work of the main thread and not to interfere in main thread's progress. Furthermore, this scheme attempts to minimize the time spent on synchronization events and transactional aborts. Helper threads perform operations on the queue, intruding at the same time as less as possible on main thread's work, even if they do not perform useful work. By using the underlying TM system there should exist a conflict resolution policy that favors the main thread and minimizes its transaction abort overheads.

Listing 4.2: Main thread's code.

```
1  while Q not empty do
2          u ← ExtractMin(Q);
3          done ← 0;
4          foreach v adjacent to u do
5                  sum ← d[u] + w(u, v);
6                  Begin−Transaction
7                  if d[v] > sum then
8                          DecreaseKey(Q, v, sum);
9                          d[v] ← sum;
10                         p[v] ← u;
11                 End−Transaction
12         end
13         Begin−Transaction
14         done ← 1;
```

```
15          End−Transaction
16  end
```

Listing 4.3: Helper threads' code.

```
1   while Q not empty do
2           while done = 1 do ;
3           x ← ReadMin(Q, tid);
4           stop ← 0;
5           foreach y adjacent to x and while stop = 0 do
6                   Begin−Transaction
7                   if done = 0 then
8                           sum ← d[x] + w(x, y);
9                           if d[y] > sum then
10                                  DecreaseKey(Q, y, sum);
11                                  d[y] ← sum;
12                                  p[y] ← x;
13                  else
14                          stop ← 1;
15                  End−Transaction
16          end
17  end
```

4.3.3 Optimizations

The authors of the papers [23] and [24] evaluated this algorithm in a full-system simulation. On the contrary, we evaluated the proposed algorithm in a real HTM system (Intel's Haswell HTM). Therefore, in order to achieve performance speedup as the number of cores increases we have applied some optimizations on the algorithm. More specifically:

- Since the basic characteristic of real HTM systems is strong isolation, there is no need to set "done" variable within a separate transaction (lines 13-15, listing 4.2). In strong isolation a conflict can be detected even if the conflicting access occurs in non-transactional code. Thus, removning this transaction (line 42 listing 4.4) we reduce the number of transactions and avoid additional cost of unnecessary transactions.

- Instead of "stop" variable used in helper threads' code to exit the inner (for) loop, we explicit abort the transaction in helper threads when "done" variable is set to 1. In this case, helper threads do not perform any other relaxations as they are explicitly aborted with an abort code that indicates this reason.

- Our implementation employs a binary heap (array representation of heap) for the priority queue. A binary heap is a complete binary search tree. All levels of the

tree except possibly the last one are full filled, and, if the last level of the tree is not complete the nodes of that level are filled from left to right. It also satisfies the min-heap ordering property, which states that the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root. Since the heap is a complete binary search tree, a heap with n nodes has $\mathcal{O}(\log n)$ height. As a result, the ReadMin function takes a constant time ($\mathcal{O}(1)$) and the ExtractMin() and DecreaseKey() functions take $\mathcal{O}(\log n)$ time.

- In an attempt to increase performance speedup we implemented a more coarse-grained transaction for the main thread. In the original algorithm the main thread performs one transaction for each edge examined (possible relaxation). This results to many small transactions, especially in dense graphs and as a consequence to additional overhead associated with the beginning and ending of many consecutive transactions. As a result, we examine more than one edge (perform possible relaxations a certain number of neighbors) within a separate transaction (line 21 listing 4.4). We expect that this coarse-grained approach is able to provide better performance speedup in small graphs and may result to more capacity aborts in large graphs. Therefore, we have to find a solution that trades off between the overhead of performing many transactions and the cost of many transactional (capacity) aborts.

- As explained in previous chapter false sharing is a liming factor for scalability. Different threads may modify independent parts of the structure that share the same cache line. Since real HTM systems detect conflicts at cache line granularity, the case of different threads that perform update operations (in transactional mode) in independent data that share the same cache line will cause data conflicts and one or more transactions will fail (abort). To avoid such conflicts and transactional aborts we applied structure padding to all shared structures like the priority queue, the distance array and the predecessor array, such as different elements of these structures to reside on different cache lines.

- Finally, the real HTM system used in evaluation part is a best effort implementation. Consequently, a fallback path is necessary. We employ a global lock, shared among threads, to protect the critical section. However, when a thread acquires the global lock, the rest will be aborted, as they have the global lock in their read set, and they will continue to fail until the lock is released. The purpose of the algorithm is to take advantage of the concept of helper threads such that to reduce main thread's relaxations and not to delay its progress. So, if a helper thread acquires the global lock, the main thread will always fail (transactional abort) until the release of the global lock and will not progress. The main thread should be allowed to run almost at the speed of the serial execution. To implement a policy that favors the main thread the global lock can only be acquired by the main thread. Helper threads always attempt to perform updates (relaxations) in the shared data through transactions and may always fail to commit, as no forward progress is guaranteed.

The listings 4.4 and 4.5 present our implementations in C programming language for the main and helper threads, respectively.

Listing 4.4: Main thread's code for a real HTM.

```
1   while(heap->curr_size > 0){
2
3           my_min = bh_extract_min(heap);
4           done = 0;
5
6           /* Find the id of the vertex. */
7           my_min_id = my_min->vertex_id;
8
9           /* Read the key (weight) of my vertex.*/
10          my_min_key = dist[my_min_id].value;
11
12          if(my_min_key < INFINITY){
13
14                  /* adjacency list for neighbors */
15                  v = g->adj[my_min_id];
16
17                  if(v != NULL){
18                          while(1){
19
20                                  /* Check neighbors for relaxation. */
21                                  begin_transaction(num_retries, &fallback_lock, tid);
22                                  for(i=0; i< num_neighbr; i++){
23
24                                          distv = dist[v->id].value;
25                                          sum = my_min_key + v->weight;
26
27                                          /* Relax */
28                                          if(distv > sum){
29                                                  decrease_key_mt(heap, v->id, sum);
30                                                  pred[v->id].value = my_min_id;
31                                                  dist[v->id].value = sum;
32                                          }
33                                          v = v->next;
34                                          if(v == NULL)
35                                                  break;
36                                  }
37                                  end_transaction(&fallback_lock, counter);
38                                  if(v == NULL)
39                                          break;
40                          }
41                  }
42                  done=1;
43          }
44  }
```

Listing 4.5: Helper threads' code for a real HTM.

```
1   while(heap->curr_size > 0){
2
3           while(done == 1);
4
5           /* ReadMin */
6           my_min_id = heap->node_array[tid].vertex_id;
7           my_min_key = dist[my_min_id].value;
8
9           if(my_min_key < INFINITY){
```

```
10
11                  /* Check neighbors for relaxation. */
12                  for(v=g->adj[my_min_id]; v!=NULL && !done; v=v->next){
13
14
15                          if(begin_transaction(num_retries, &fallback_lock, tid) != -1){
16                                  if(done == 0){
17                                          distv = dist[v->id].value;
18                                          sum = my_min_key + v->weight;
19
20                                          if(distv > sum){
21                                                  decrease_key_mt(heap, v->id, sum);
22                                                  pred[v->id].value = my_min_id;
23                                                  dist[v->id].value = sum;
24                                          }
25                                  }else
26                                          _xabort(0xaa);
27                          }else
28                                  break;
29                          end_transaction(&fallback_lock, counter);
30
31                  }
32          }
33  }
```

4.4 System Configuration

The system we used to evaluate the proposed parallelization of Dijkstra's algorithm was a 28-core platform (Figure 4.5), NUMA architecture with the following characteristics.

- 2 sockets (Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz)
- 14 cores per socket (28 threads with hyperthreading)
- 32KB L1 data cache per core
- 32KB L1 instruction cache per core
- 256KB L2 cache per core
- 35MB L3 cache per socket
- 128GB RAM
- Hardware Transactional Memory:
  - lazy data versioning
  - eager conflict resolution
  - best effort HTM
  - strong isolation
  - cache line granularity
  - 4MB read set
  - 22KB write set

In the evaluation part, each software thread is manually pinned to a hardware thread (to a core) in order to take advantage of the locality with the sockets. We first pin software

Figure 4.5: The platform used in evaluation of Dijkstra's algorithm.

threads such that to fill the first socket (first 14 threads) and share the same L3 cache. And then we pin threads in the second socket. Our evaluation reveals that the NUMA effect negatively influences the scalability. In case of a cache miss the transfer of the memory address from one socket to another is costly.

4.5 Experimentation

4.5.1 Experimentation in the serial algorithm

We first evaluated the serial Dijkstra's algorithm. More specifically, we executed on the above platform the main thread's code (simple dijkstra's algorithm) for graphs of different sizes, both dense and sparse graphs. In this experimentation there is no need to perform relaxations within a transaction, as this is a serial execution. Executing relaxations in transactional mode causes an additional overhead associated with the transaction and it is more time consuming. However, the difference in the runtime between using and not using transaction is not important and does not affect our analysis. The main purpose of this experimentation is to evaluate the parallelization of the inherently serial Dijkstra's algorithm.

We separated the algorithm in four phases and measured the runtime of each phase. The first phase is the ExtractMin() operation that takes time proportional to $\mathcal{O}(\log n)$, the second phase is the compute operation for the distance (line 9 in listing 4.6) from

source, the third is the DecreaseKey() operation and the fourth is the update operation in the distance and predecessor arrays (lines 18-19 in listing 4.6). Figure 4.6 presents our results for two dense graphs (a rmat graph with 1M nodes and 100M edges, and a random graph with 10M nodes and 500M edges) and a sparse graph (a rmat graph with 100M nodes and 100M edges).

Listing 4.6: The four phases of the algorithm.

```
1  while Q not empty do
2          start_timer(extract_min)
3          u ← ExtractMin(Q);
4          stop_timer(extract_min)
5
6          done ← 0;
7          foreach v adjacent to u do
8                  start_timer(compute_time)
9                  sum ← d[u] + w(u, v);
10                 stop_timer(compute_time)
11
12                 Begin−Transaction
13                 if d[v] > sum then
14                         start_timer(decrease_key)
15                         DecreaseKey(Q, v, sum);
16                         stop_timer(decrease_key)
17                         start_timer(update_time)
18                         d[v] ← sum;
19                         p[v] ← u;
20                         stop_timer(update_time)
21                 End−Transaction
22         end
23
24         Begin−Transaction
25         done ← 1;
26         End−Transaction
27 end
```

Taking into consideration the concept of this parallel algorithm, only the decrease_key and update part of the algorithm can be offloaded from the main thread. Helper threads perform some operations such that the if branch of the main thread (line 13 in listing 4.6) can be evaluated as true fewer times. According to figure 4.6 this branch constitutes almost the 12-25% of the total main thread's runtime. Thus, the main thread can gain a very small percentage of the total runtime and this parallelization can offer small performance speedups theoretically. The extract_min and compute phases of the algorithm have to be executed from the main thread too, for all the nodes and the edges of the graph. As a result, we conclude that dijkstra's algorithm is a hard algorithm to parallelize as the most part of

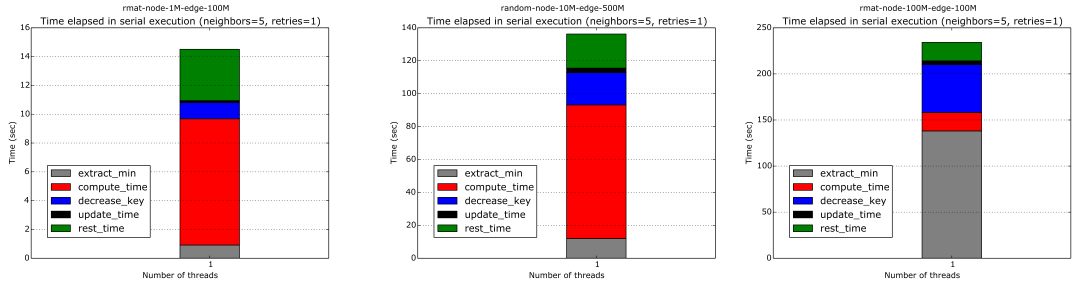


Figure 4.6: Evaluation of the four phases of the algorithm in different graphs.

the algorithm is serial.

In the rmat-node-100M-edge-100M graph the extract_min phase constitutes a great portion of the total runtime, comparing with the other two graphs. ExtractMin() in a binary heap removes the root of the heap and replace it with the node with the next highest priority (minimum distance from the source). To implement this a traversal from root to leaves is needed and it takes time proportional to $\mathcal{O}(\log n)$. Thus, the more nodes a graph has, the longer the ExtractMin() function lasts. This conclusion is depicted in rmat-node-100M-edge-100M graph that is a very large graph. We can also conclude the same for the DecreaseKey() function, as it also takes time proportional to $\mathcal{O}(\log n)$.

On the other hand, in the other two graphs (rmat-node-10M-edge-500M and random-node-1M-edge-100M) the compute phase is the most time-consuming phase of the algorithm. Compute phase calculates a new distance associated with an edge. It repeats this calculation for all edges in the graph. As a result, the more dense a graph is, the more time the compute phase lasts. The rmat-node-100M-edge-100M is a sparse graph, as the ratio of the number of nodes to the number of edges is 1. In this graph, every repetition of the while loop (line 1 in listing 4.6) demands only one repetition on average of the compute phase, while in more dense graphs the compute phase is repeated many times. Therefore, this phase is the most time-consuming phase in dense graphs and does not constitute a large portion of the total time in sparse graphs.

In the serial execution there is no data conflict aborts, since there is only one thread. We used structure padding technique in order to avoid false sharing that would lead to data conflicts aborts in case of multiple threads in the execution. Thus, the structures of the algorithm like the distance and predecessor arrays have not to be padded in the serial execution. We evaluated the serial algorithm for different graphs without using structure padding technique in the shared structures and our results are shown in figure 4.7.

Our experiments demonstrate that although we do not use padding in the structures, the proportion of the 4 phases of the algorithm (extract_min, compute, decrease_key and update) to the total runtime remains the same. The pattern of the different phases is similar in all executions. However, we have to notice that the total runtime is reduced significantly when we remove the padding from the distance array in executions where the compute phase is the most time-consuming phase (dense graphs). Without padding in the distance array, consecutive elements are stored in the same cache line. Therefore, the main thread

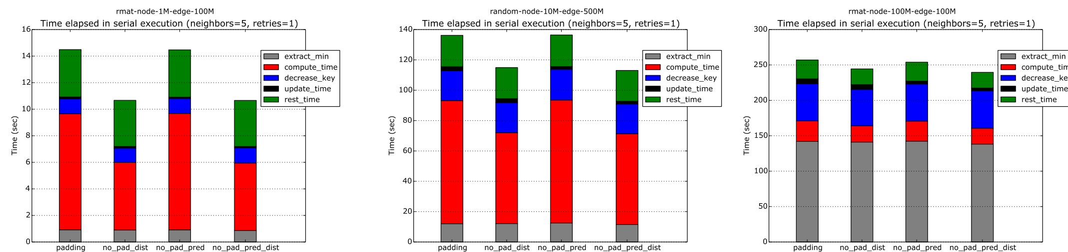


Figure 4.7: Experimentation in padding technique on the serial algorithm.

can find some elements in its cache memory and avoid transferring cache lines from the main memory in each read/write operation. Furthermore, without structure padding the capacity misses are reduced. One cache line can store multiple elements of the array and cache memory can store more elements than before (in case of padding).

As a consequence, the main gain of removing padding is in compute phase, where we avoid transferring the distance value of the current node d[u] (line 9 in listing 4.6) in each iteration of the for loop (line 7 in listing 4.6). This element remains in the cache memory and the main thread can read its value quickly. In sparse graphs like rmat-node-100M-edge-100M graph (figure 4.7) there is not considerable gain, as the compute phase is not so time-consuming. Finally, removing padding from the predecessor array (p[]) does not have any important contribution in the total runtime. In each iteration of the for loop the adjacent nodes (v nodes) of the current u node can be in the same cache line of the predecessor array with a very small probability. It depends on the shape of the graph. In the most common case, edges connect nodes which are quite remote one from each other (not consecutive nodes). These nodes are not in the same cache line and as a result the main thread has to transfer the element $p[v]$ from the main memory every time that if branch is evaluated as true despite not using padding. It cannot exploit spatial locality.

4.5.2 Experimentation in the parallel algorithm

Experimentation in the number of retries per transaction before acquiring the lock

As explained in previous chapter, when a transaction fails to commit, it retries. Nevertheless, there are transactions that they always fail to commit. For example, when transaction's read/write set exceeds HTM system's read/write set, the transaction will be always aborted due to capacity abort. In this case the fallback path must be executed. In our scheme the fallback path is implemented as coarse-grained locking code that uses a global lock shared among all threads that protects the critical section. If a thread acquires this global lock, any other thread cannot proceed in the critical section and it will always fail until the lock is released.

We evaluated the performance of the algorithm for different numbers of retries for a transaction before acquiring the global lock. First, the main thread performs a certain

number of retries for each transaction and if it exceeds this number due to consecutive transactional aborts, it acquires the lock, aborts all helper threads and executes the critical section in a coarse-grained locking mode. Helper threads can never acquire the lock and they always attempt to execute the critical section in transactional mode (unlimited number of retries for a transaction). In the opposite case, where helper threads could acquire the global lock, they would delay the main thread and would interfere its progress decreasing the performance of the algorithm. Figures 4.8 and 4.9 present a performance evaluation for different numbers of retries for a main thread's transaction (a random-node-1M-edge-10M and a rmat-node-10M-edge-500M respectively) investigating 5 neighbors for relaxation per transaction.

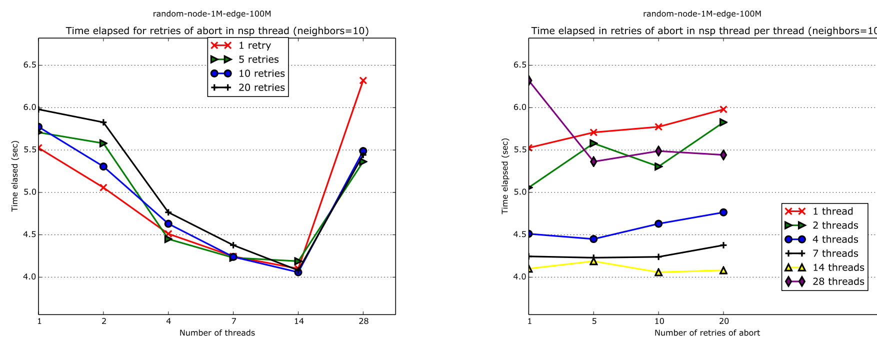


Figure 4.8: Time elapsed in random node-1M-edge-10M graph for different number of possible retries per transaction of the main thread.

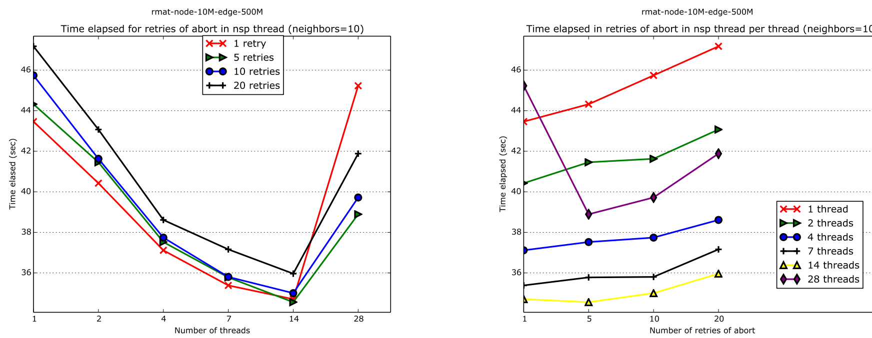


Figure 4.9: Time elapsed in rmat node-10M-edge-500M graph for different number of possible retries per transaction of the main thread.

According to these figures, the more times a transaction can retry, the more time the execution of the algorithm lasts. In case of one possible retry per transaction, when the main thread fails to commit, it immediately acquires the lock and executes the critical section successfully while helper threads are stalled. Thus, this execution of the main thread is very close to the serial execution. Since the main thread can retry its transaction

more times (the number of retries per transaction increases), each transaction lasts more when consecutive transactional aborts appear. This is why the runtime is better in the execution with one possible retry per transaction comparing with the rest executions with more possible retries per transaction, where the main thread's runtime diverge more and more from the serial execution's runtime.

Secondly, we can notice that in executions with more than one possible retry there is a better scalability. For example, the runtime of 14 threads in executions of 5 and 10 retries per transaction becomes approximately the same with that of one possible retry per transaction (figure 4.9), while in case of one thread they differ significantly. This is because in executions of one possible retry per transaction helper threads are stalled immediately without performing many relaxations. In executions with more possible retries per transaction helper threads have more time to perform relaxations and commit them, so as the main thread can gain more work.

Finally, in executions with 28 threads it appears the effect of NUMA architecture. The transfer of a cache line from the memory of one socket to the another is costly and it negatively influences the scalability. The runtime increases because of expensive transfers of cache lines from the memory of a different socket. We can also observe that the time elapsed of the execution with one possible retry per transaction is worse than in executions with more possible retries per transaction. This is due to the shared global lock. The main thread writes (locks) the global lock and each time that a helper thread starts a transaction and attempts to read the global lock, it has to transfer it (because of the coherence protocol) possibly from a remote memory. This is quite costly in executions where the main thread writes the global lock frequently like in execution with one possible retry per transaction. In this execution when a helper thread, which resides in different socket from the main thread, reads the global lock, it has to transfer it from a remote memory because it has been written from the main thread with some strong probability. On the other hand, in executions with more possible retries per transaction the main thread does not write the global lock so frequently and thus, helper threads do not perform costly transfers of the global lock in each transaction (frequently). However, each transaction lasts more when consecutive transactional aborts show up. To sum up, in case of 28 threads where 2 sockets are used, we have to find a solution about the number of possible retries per transaction that trades off the costly frequent transfers of the global lock and the more time-consuming transactions in case of consecutive transactional aborts in the main thread.

Subsequently, we examined the number of possible retries per transaction in helper threads. In previous executions helper threads could retry their transaction until they commit or are explicitly aborted by the main thread when "done" variable is set to 1. We attempted to limit the number of possible retries per transaction in helper threads, such that to reduce conflict aborts of the main thread. Helper threads can now perform a specific number of retries per transaction and if they exceed this number they will wait in a while loop until they are forced by the main thread to proceed in the next iteration (until "done" is set to 1). We suppose that retrying a transaction to commit after a certain number of retries can only cause conflict aborts and can never lead to a transactional commit. Figures 4.10 and 4.11 depict our results for different numbers of possible retries per transaction in

helper threads.

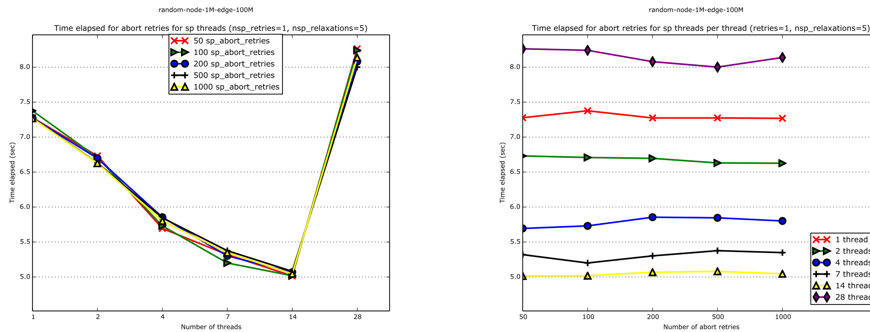


Figure 4.10: Time elapsed in random node-1M-edge-10M graph for different number of possible retries per transaction of helper threads.

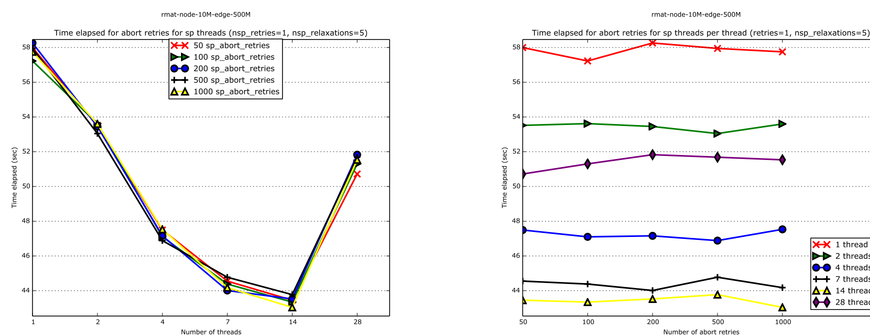


Figure 4.11: Time elapsed in rmat node-10M-edge-500M graph for different number of possible retries per transaction of helper threads.

Our results demonstrate that although we limited the number of possible retries per transaction, the total runtime of the algorithm was not improved. Helper threads can commit their transactions retrying them less times than the given number (limit) of retries. Even though we reduced the number of possible retries per transaction to a very small number, 50 retries, this number is like an infinite value for retries. Thus, we conclude that there is no gain by limiting the number of retries per transaction in helper threads.

Experimentation in the number of neighbors examined for relaxation per transaction

To avoid the overhead of beginning and ending many consecutive small transactions, we implemented a more coarse-grained scheme in transactions for checking the neighbors of the current node to perform relaxations on them. If we check more than one neighbor to relax in a single transaction, the overhead of performing transactions reduces, as we

have bigger and fewer transactions. However, in large graphs performing a more coarse-grained transaction can lead to capacity transactional aborts, since this scheme accesses a large part of the memory that can exceed HTM system's read or write set. Therefore, we examined the number of neighbors checked for relaxation within a single transaction, such that to trade off the overhead of many consecutive small transactions and the capacity transactional aborts that may appear in a more coarse-grained scheme.

We evaluated the algorithm as described in listings 4.4 and 4.5 for the main and helper threads, respectively. More specifically, we performed a coarse-grained transaction for different number of neighbors to examine in the main thread, while helper threads check only one neighbor for relaxation in each transaction. We executed the algorithm in different sizes of graphs for 1, 2, 5, 10, 20 and 50 neghbors examined for relaxation in a single transaction. Figures 4.12 and 4.13 demonstrate our results for a random node-1M-edge-100M graph and a rmat node-10M-edge-500M graph.

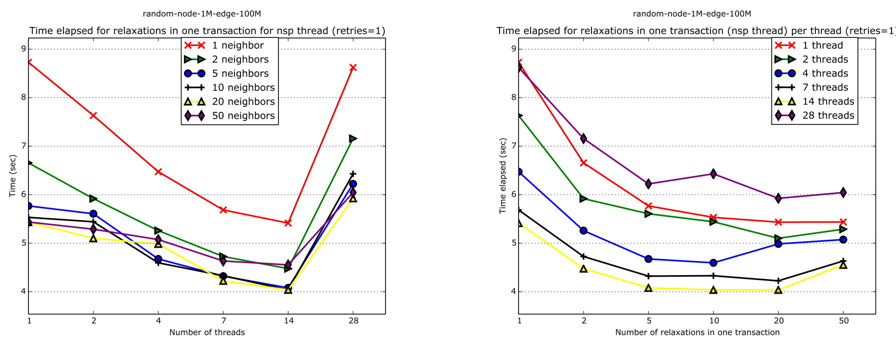


Figure 4.12: Time elapsed in random node-1M-edge-100M graph for different number of neighbors examined for relaxation per transaction of the main thread.

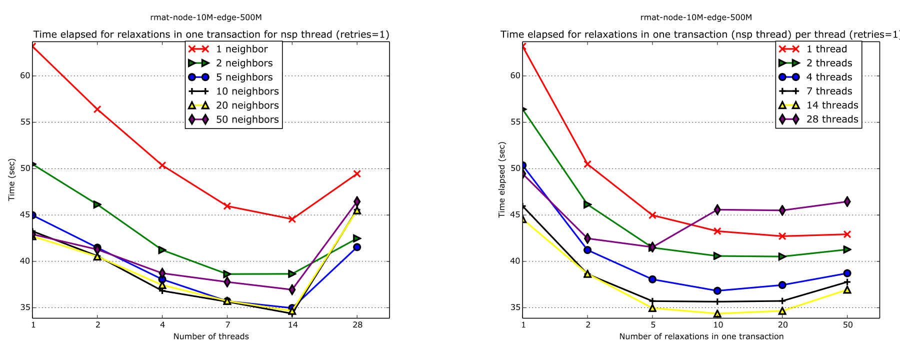


Figure 4.13: Time elapsed in rmat node-10M-edge-500M graph for different number of neighbors examined for relaxation per transaction of the main thread.

Our results confirm our hypothesis about a more coarse-grained transaction. We can observe that the case of checking one neighbor per transaction has the worse total time

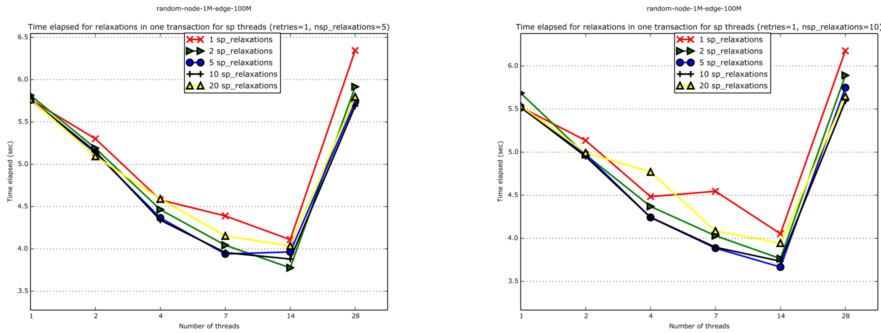


Figure 4.14: Time elapsed in random node-1M-edge-100M graph for different number of neighbors examined for relaxation per transaction of helper threads.

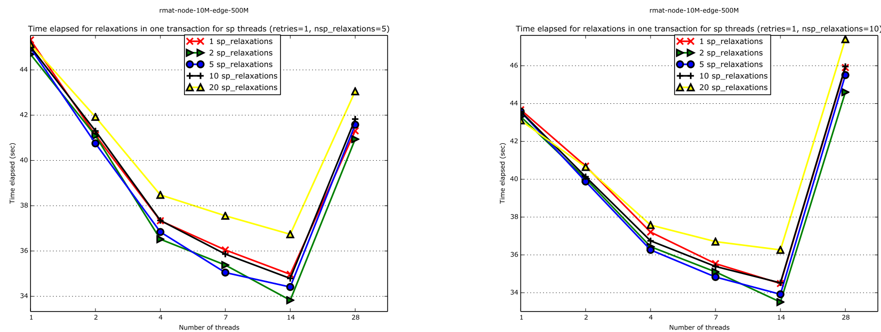


Figure 4.15: Time elapsed in rmat node-10M-edge-500M graph for different number of neighbors examined for relaxation per transaction of helper threads.

elapsed, since there are many small consecutive transactions that clear and fill frequently transactional cache lines in memory and this is quite time-consuming (high overhead). As the numbers of neighbors checked per transaction increases the total time elapsed is improved until a certain number of neighbors. In the smaller graph (random-node-1M-edge-100M figure 4.12) the best time elapsed can be shown for 20 neighbors checked within a single transaction, while in the larger graph (rmat-node-10M-edge-500M figure 4.13) the best time elapsed is appeared in case of 10 neighbors. Consequently, we can conclude that the larger a graph is, the less coarse-grained the transaction should be, since more coarse-grained transactions in large graphs access more memory (the priority queue is larger) and they may lead to capacity transactional aborts.

Secondly, the case of checking one neighbor per transaction scales better as the number of threads increases. In this execution, the time elapsed for one thread only (the main thread) is the worst and thus, the more threads we add, the more gain we have. The executions with more than one neighbor checked for relaxation per transaction are not so time-consuming and adding more threads do not have so much gain (smaller scalability) than in case of examining one neighbor per transaction. Finally, executions with 28 threads

are time-consuming because of the NUMA architecture effect of our system which causes costly cache line transfers from one socket to another.

In the next step, we performed the same evaluation for helper threads, too. We fixed the coarse-grained transaction of the main thread to 5 and then to 10 neighbors per transaction and implemented a coarse-grained transaction for helper threads (1, 2, 5, 10, 20 neighbors examined for relaxation per transaction). Our purpose is to analyze which execution has the best scalability, since all executions start from the same point, the one thread execution (main thread) with fixed (5 or 10) neighbors checked per transaction. Our results are depicted in figures 4.14 and 4.15.

In random-node-1M-edge-100M graph the executions of 5 and 10 neghbors examined per transaction have the best scalability. Similarly to the above conclusions, as the transaction becomes more coarse-grained (bigger transactions), we can avoid costly consecutive small transactions, but we do not gain in performance if we exceed a certain number of neighbors. In rmat-node-10M-edge-500M graph the executions of 2 and 5 neighbors checked per transaction give the best scalability. Since this graph is larger, the transaction has to be less coarse-grained than in the smaller graph. It accesses a larger binary heap and can exceed HTM system's read/write set by checking for relaxations a smaller number of neighbors within a single transaction.

4.6 Results

4.6.1 Performance results

In our performance evaluation we tested graphs of different density and structure. We used graphs with 10K, 1M, 10M, 100M vertices from the Random and R-MAT families. We also evaluated the algorithm on a real road network, a full USA road network (USA-road-d.USA). Figure 4.16 presents the speedups achieved by the implementation of Dijkstra's algorithm described in listings 4.4 and 4.5. The speedup obtained for n threads is the ratio of the execution time of the serial algorithm to the execution time with n threads, $n-1$ of them being helper threads. This scheme is able to achieve significant speedups in most cases. The maximum speedup achieved is 1.39 for the random-node-1M-edge-100M graph (14 threads).

As explained in [24] in the serial execution, time can be estimated as:

$$T_{serial} = n * \mathcal{O}(\log n) + d * n * \mathcal{O}(\log n), \tag{4.4}$$

where n represents the number of vertices in the graph and d the average out-degree of the vertices. The ExtractMin() operation spends time $n * \mathcal{O}(\log n)$ and DecreaseKey spends $d * n * \mathcal{O}(\log n)$ time, approximately.

The execution time of the described parallel scheme can be estimated as:

$$Tparallel = n * \mathcal{O}(\log n) + a * d * n * \mathcal{O}(\log n), a < 1 \tag{4.5}$$

where a is the ratio of the main thread's DecreaseKey() operations to those executed in the serial case. This is a simple theoretical approach. It does not take into account the time

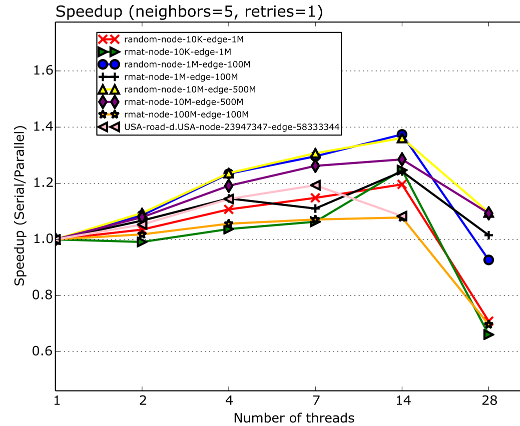


Figure 4.16: Multithreaded speedups for graphs of different density.

spent for thread coordination or delays due to transactional aborts. Thus, a theoretical speedup can be calculated as:

$$s = \frac{1+d}{1+a*d} \tag{4.6}$$

We have to notice that the performance is strongly related to the density of the graph. This conclusion is implied to speedup's definition 4.6, too. According to the results of figure 4.16, for more dense graphs, the speedup is greater, as more parallelism can be exposed in the inner loop of the algorithm. Conversely, sparse graphs like rmat-node-100M-edge-100M leave limited space for parallelism leading to low performance. Furthermore, the figure also reveals that the speedup increases as more threads are utilized. The performance is improved up to a maximum point, after which utilizing more threads leads to performance degradation. The number of threads to achieve this maximum is again related to the graph's density. For example, in rmat-node-100M-edge-100M (sparse) graph increasing the number of threads from 7 to 14 slightly reduces the performance and in USA-road-d.USA-node-23947347-edge-58333344 graph the performance remains nearly the same. Finally, using 28 threads degrades the performance in all evaluated graphs because of the NUMA effect. Our system is a NUMA architecture with 14 threads per socket. Therefore, pinning 28 threads in two sockets leads to expensive cache line transfers from one socket to another and negatively influences scalability.

4.6.2 A closer look at the results

In this subsection, we attempted to have a closer look into the behavior of the described scheme. We examined main thread's gain in the number of relaxations, the abort ratio and the time spent in each main thread's operation and we tested them to all previous graphs.

We only present some representative graphs with different density, as the other graphs exhibit similar behavior.

Figure 4.17 shows the distribution of performing relaxations between the main and helper threads (lines 28-32 and 20-24 in listings 4.4 and 4.5 respectively). As more threads are used, main thread's relaxations are reduced and helper threads' relaxations are increased, justifying the performance improvement. Similar reductions in the main thread's operations are also achieved for the sparse graph (rmat-node-100M-edge-100M). In this graph, helper threads' relaxations can never exceed main thread's relaxations, as this is a very sparse graph and helper threads cannot offload many relaxations of the main thread. Employing 28 threads degrades scalability of main thread's relaxations because of the NUMA effect, that was depicted in the previous figure 4.16 for speedup, too.

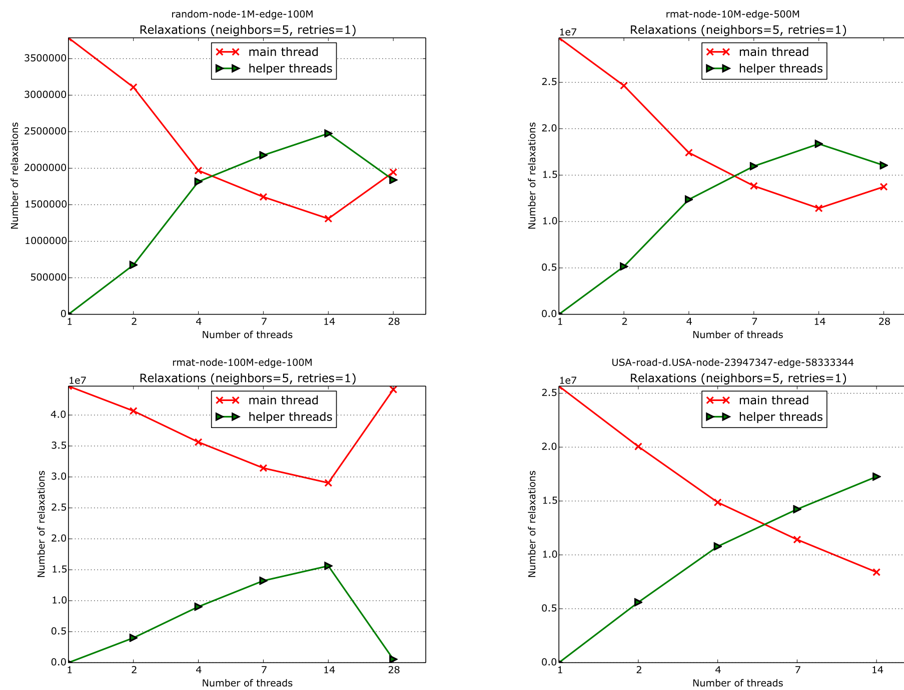


Figure 4.17: Distribution of relaxations between the main and helper threads.

Figure 4.18 depicts the number of commits and aborts of the main thread for the evaluated executions. The main thread suffers a really low number of aborts, especially in dense graphs. This means that even when helper threads are not contributing any useful work, they still do not obstruct main thread's progress. The main thread is allowed to run almost at the speed of the serial execution. An important observation though, is that the number of transactional aborts in the main thread depends on the size of the transaction's write set. The larger the write set is, the higher probability of a conflict. Furthermore, in the more coarse-grained transaction that we have implemented for the main thread, there is a higher probability of capacity aborts in large graphs like the full USA road network, where in main thread's execution there are many transactional capacity aborts. Finally, the

addition of more threads does not lead to an increase of the number of aborts. Thus, we can suppose that if the NUMA effect did not exist, the algorithm would lead to a better performance speedup for more than 14 threads.

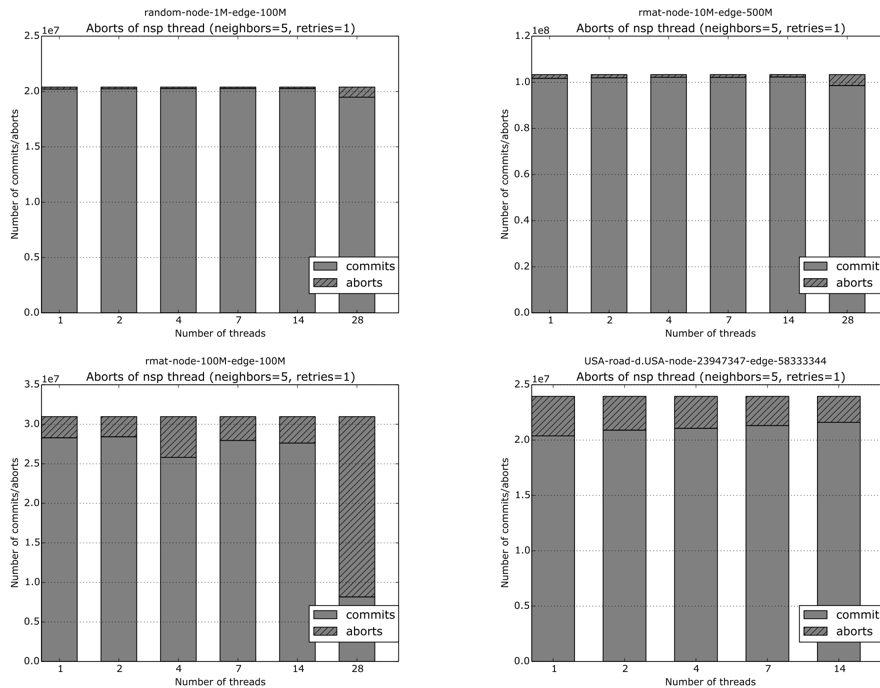


Figure 4.18: The number of main thread's commits/aborts.

In case of 28 threads transactional aborts are increasing. A single transaction lasts more time due to the NUMA effect. The cache line transfers are expensive (take a long time), since cache lines may be transferred from a remote memory. As the transaction is more time-consuming, it can be aborted with a stronger probability. Firstly, data conflicts are more possible to be detected in longer transactions. And secondly, if a transaction last more than the time quantum, the scheduler of the operating system will schedule out the process and the transaction will be aborted because of a timer interrupt. In rmat-node-100M-edge-100M graph, the transactional aborts in 28 threads due to the NUMA effect are significantly increased, since this is the largest graph and a single transaction takes up a lot of time.

To gain a better understanding of the wasted work due to transactional aborts, figure 4.19 presents the percentage of the total transactional aborts for all threads in total number of transactions. Again, for graphs of high density the percentage of transactional aborts is relatively small (about 3%), justifying the observed speedups, while in sparse graphs this percentage is higher. The small percentage of transactional aborts shows that most of the concurrent accesses to the shared data structures are non-conflicting. Moreover, we can notice that in dense graphs, as the number of threads increases, the percentage of transactional aborts also increases, as the probability of performing conflicting accesses

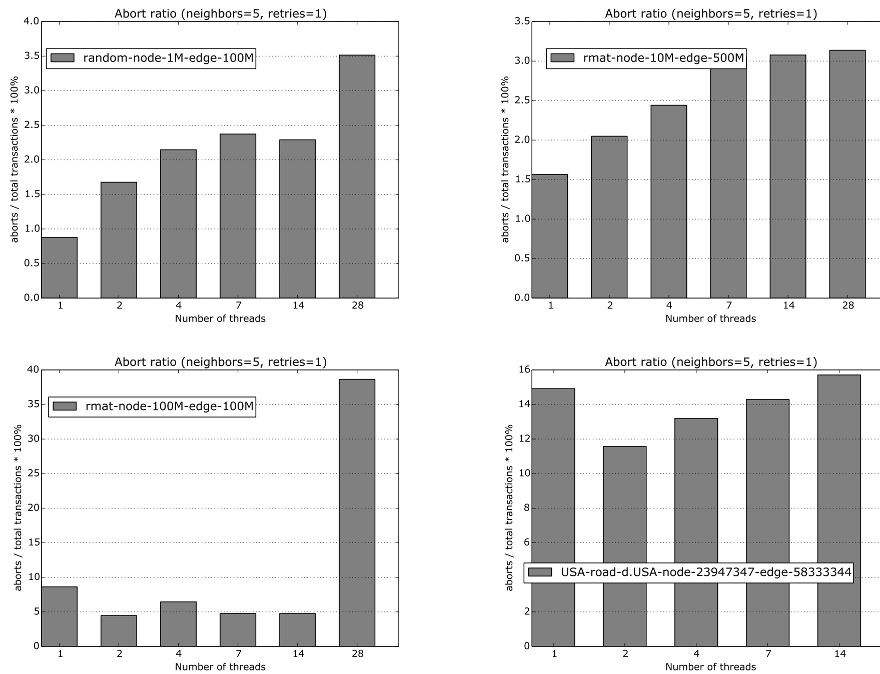


Figure 4.19: The percentage of total transactional aborts for all threads in total number of transactions.

becomes stronger. On the contrary, in sparse graphs like the rmat-node-100M-edge-100M graph the number of transactional aborts does not augment when adding more threads because threads perform conflicting accesses with a very small probability in sparse graphs. Similarly to figure 4.18, the percentage of transactional aborts is increased in 28 threads, since the transaction lasts longer due to the NUMA effect and becomes more vulnerable to a transactional abort.

Figure 4.20 shows the time spent by the main thread in ExtractMin() operation, in transactional mode part (lines 21-37 in listing 4.4) and in the rest operations of the main thread. The ExtractMin() operation remains stable for each graph, as it is not affected in the described scheme. The time spent in ExtractMin() is only increased in case of 28 threads because of the costly cache line transfers in our NUMA architecture. The NUMA effect is notably depicted in the large graph (rmat-node-100M-edge-100M graph). Secondly, the addition of helper threads reduces the time spent in transactions, the parallel part of the scheme. The main thread performs less relaxations (lines 28-32 in listing 4.4). It executes fewer times the costly DecreaseKey() operation, that takes time proportional to $\mathcal{O}(\log n)$ (where n is the number of vertices) and as a result the runtime in transactional mode reduces. Finally, as explained in the evaluation part of the serial execution, the ExtractMin() operation lasts more time and constitutes a large percentage of the total runtime in larger graphs like rmat-node-100M-edge-100M graph and USA-road-d.USA graph, since it also takes time proportional to graph's size ($\mathcal{O}(\log n)$).

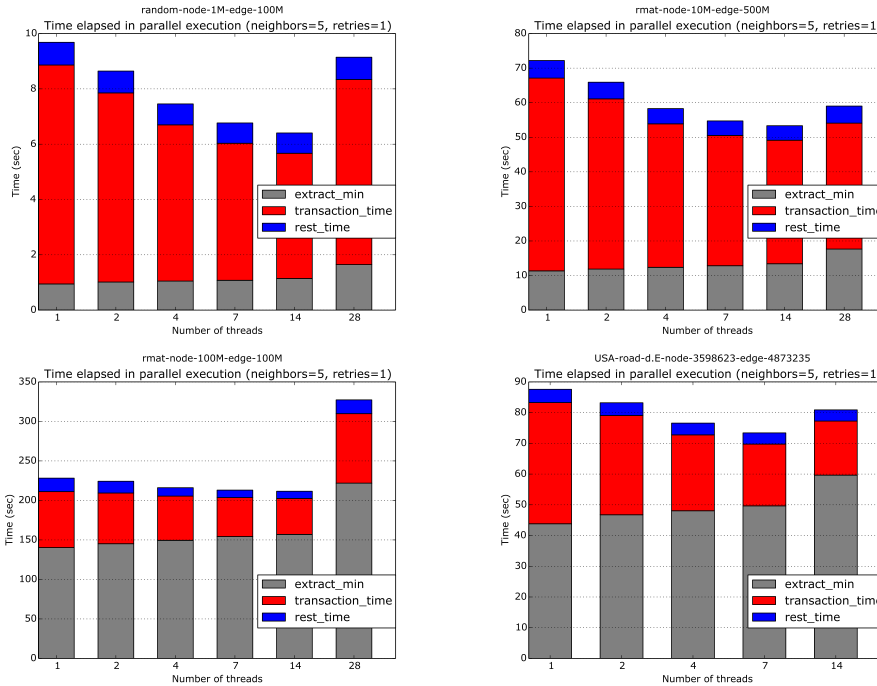


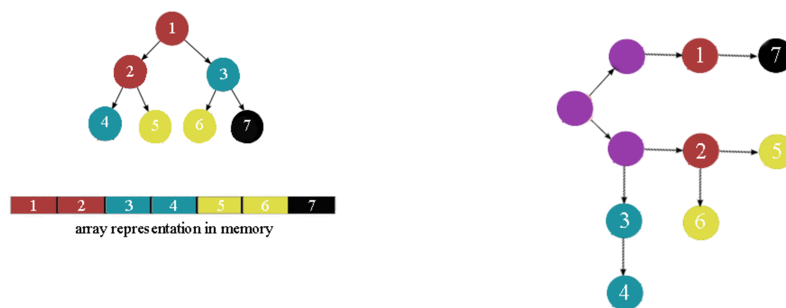
Figure 4.20: Distribution of time spent in different phases of main thread's execution.

4.6.3 Experimentation in padding technique

All previous evaluations were implemented using padding technique in shared structures. We padded the distance and predecessor arrays as well as the node_array and the where_in_heap array, which are used for the graph representation and are shared among threads, too. We supposed that if we were not using padding technique, the number of transactional aborts would be higher and thus, the execution in transactional mode would take a longer time. Different threads that perform operations (in transactional mode) in independent data that reside in the same cache line would be aborted, as our real HTM system detects conflicts at cache line granularity. However, without using padding memory can store more elements of the structures. As a consequence, a thread can find an element in its cache memory with a stronger probability, avoid an expensive transfer and can exploit temporal and spatial locality. Temporal locality defines that a data which is referenced at one point in time will be referenced again sometime in the near future and spatial locality defines that likelihood of referencing a data is higher if a data near it was just referenced.

In order to verify our hypothesis for longer time spent in transactional mode in case of not using padding technique, we executed the algorithm removing padding of the shared structures. Our results can be shown in figure 4.22. They prove that our hypothesis was incorrect, as the transactional runtime was reduced. Different threads access independent

data that reside in the same cache line with a very small probability. It depends on the shape of the graph. In the most common case, edges connect vertices that are quite remote one from each other (not consecutive nodes) and as a result they do not share the same cache line. Thus, we can conclude that not using padding does not affect the abort ratio. Figure 4.21a presents a representation of a binary heap in memory (vertices that share the same cache line are depicted with the same color) and figure 4.21b the real network in which edges connect nodes that reside in different cache lines. In case of 3 threads, the first thread will relax node 7, the second nodes 5, 6 and the third node 4. These simultaneous relaxations are performed in nodes that reside in different cache lines despite not using padding. As a result, it will not appear any transactional abort.



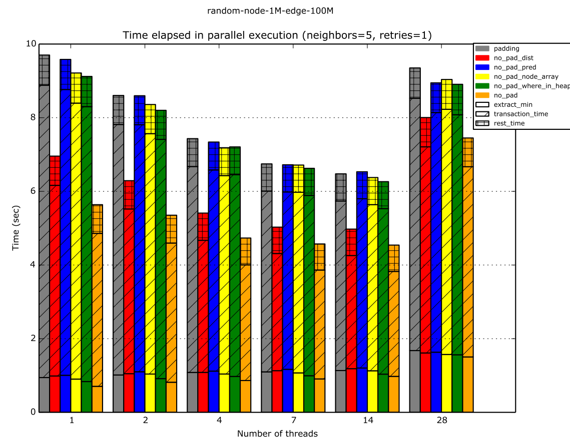
(a) Binary heap representation in memory.

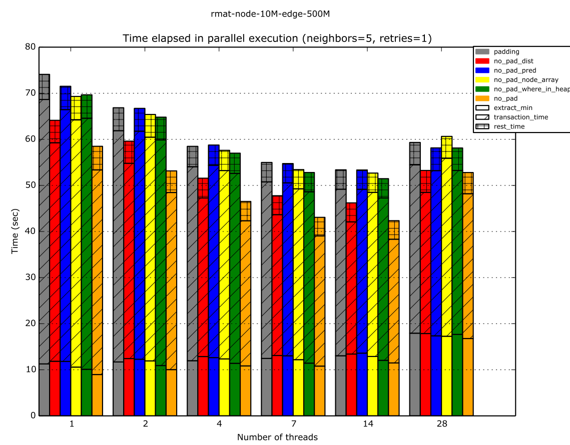
(b) The real network.

Figure 4.21: Despite not using padding, simultaneous relaxations are performed in nodes that reside in different cache lines.

On the other hand, removing padding improved the total runtime of the algorithm. More specifically, removing padding reduces the time spent in transactional mode, especially in case of removing padding from the distance array, since the distance array is accessed mostly inside transaction (line 24 in listing 4.4). The if branch (lines 28-32) that accesses the rest structures (predecessor array, node_array and where_in_heap array) is evaluated as true fewer times. Therefore, the main gain of temporal and spatial locality for a thread is in the distance array. Threads can find an element in their cache memory with a stronger probability when no padding is used and avoid transferring cache lines from the main memory. Furthermore, by avoiding frequent transfers we can also avoid delays related to the common memory bus (memory bus congestion).

Finally, we can notice that despite removing padding the execution pattern remains the same. As the number of threads increases, the scalability of the execution without using padding in structures is the same with that of using padding. The performance speedups of the algorithm have the same values for all evaluated graphs. The only difference is that the total runtime of the algorithm is reduced when no padding is used in structures. Consequently, our analysis for the scalability and the performance of the algorithm is not affected.





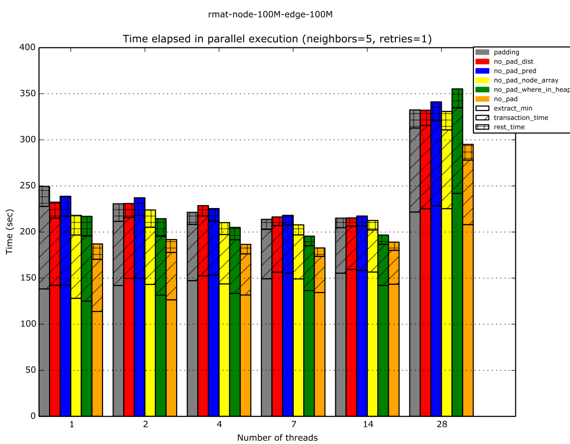


Figure 4.22: Experimentation in padding technique on the parallel algorithm.

4.7 Employing skip list

4.7.1 The skip list structure

Our implementation described employs a binary heap for the priority queue. In this section we attempted to evaluate the algorithm employing a skip list instead of binary heap. DecreaseKey() operation takes time proportional to $\mathcal{O}(\log n)$ for both skip list and binary heap and ReadMin() operation has the same complexity, too. However, ExtractMin() operation takes time proportional to $\mathcal{O}(1)$ when using a skip list, in contrast with the binary heap where it takes $\mathcal{O}(\log n)$.

Skip list is a data structure that allows fast lookup within an ordered sequence of elements (key-value pairs). Fast search is made possible by maintaining a linked hierarchy of subsequences, each skipping over fewer elements. This structure is built in layers. The bottom layer is an ordinary ordered linked list. A key in layer i appears in layer i+1 with some fixed probability p. Thus, each element of the structure has a random height that represents the layers in which the associated key appears. Skip list has a maximum height and every time that an element is inserted, it takes a random height between 1 and skip list's maximum height. Figure 4.23 depicts an example of a skip list.

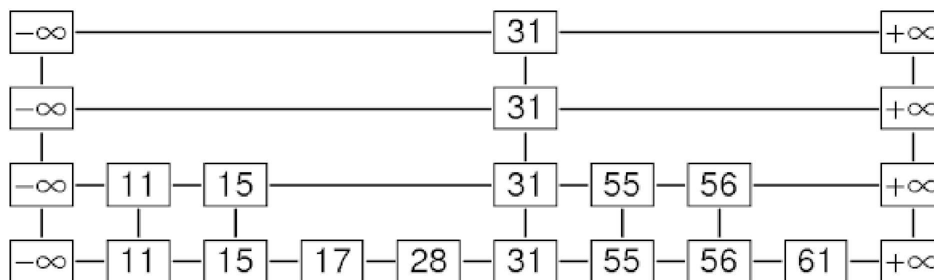


Figure 4.23: An example of a skip list.

A lookup for a target begins at the head element in the top layer and proceeds horizontally until an element greater or equal to the target is reached. If this element is equal to the target, it has been found and the operation returns. Otherwise, the procedure is repeated after returning to the previous element and dropping down vertically to the next lower layer. The total expected complexity of the lookup operation is $\mathcal{O}(\log n)$.

In skip list, the height of each element (number of element's layers) is a random number. Thus, it is possible (though with a very low probability) that it will be produced a badly balanced structure. However, skip lists work well in practice, and the randomized balancing scheme has been argued to be easier to implement than the deterministic balancing schemes used in other structures like balanced binary search trees. Finally, skip lists are useful in parallel computing, where insertions can be done in different parts of the skip list concurrently without any global rebalancing of the data structure.

4.7.2 Comparison with the binary heap

In this section we will compare the implementation that employs a skip list for the priority queue with the previous implementation (binary heap for the priority queue). In our first approach we constructed a skip list that had an element for each vertex of the graph. Thus, in DecreaseKey() operation the element that represents the vertex to be relaxed, is removed from the skip list and is placed at a closest position to the top of the list. As in case of using a binary heap, this operation takes time proportional to $\mathcal{O}(\log n)$. However, the execution time of the algorithm was extremely large and the scalability was not remarkable. The reason was the DecreaseKey() operation, which was more time-consuming than that in case of using a binary heap.

In our attempt to improve the implementation with the skip list, we noticed that the DecreaseKey() operation was performing many steps to place the extracted element to its new position. To place an element in skip list, there is a traversal from the head of the list until an element with an equal or a greater key is reached. This traversal was taking too many steps, while placing the element to its new position in the binary heap requires a small number of swaps in the elements of the structure. We also noticed that our skip list had too many elements with the same key during the execution of the algorithm. While the algorithm were executed, the distances of the vertices from the source were being updated with the same value with some strong probability. As a consequence, our skip list had many elements (different vertices of the graph) with the same key, was needing a lot of memory (more cache line transfers) and the traversal from the head of the list to the desired node, was much time-consuming (due to the large number of elements which had to be overtaken).

Therefore, we implemented an optimized skip list that contains only discrete keys. Each element of the list has a unique key and a simple internal nested list that stores the ids of the vertices which have the same distance-key from the source. In this way, our skip list demands less memory, since it contains fewer items, and the DecreaseKey() operation takes fewer steps, since it traverses a smaller number of elements.

In our evaluation we firstly compare the serial execution of the algorithm for the three different structures, the binary heap, the simple skip list (there is an element for each vertex of the graph) and the optimized skip list (it contains only discrete keys). Figure 4.24 shows our results for two different graphs, the rmat-node-10M-edge-500M and rmat-node-100M-edge-100M. The ExtractMin() operation is much less time consuming in implementations that employ a skip list, since it takes time proportional to $\mathcal{O}(1)$, while in case of binary heap it takes $\mathcal{O}(\log n)$. In large graphs like the rmat-node-100M-edge-100M graph, where the ExtractMin() operation constitutes a great portion of the total runtime, there is a significant gain. Furthermore, we have to remark that the DecreaseKey() operation takes more time in the simple skip list for both two graphs. As explained above, this is because it takes too many steps to place the extracted element to its new position, since it traverses many elements of the same key. In our optimized skip list we avoid such a long traversal and as a result the DecreaseKey() operation can place an element to its new position with approximately the same number of steps with the binary heap. The elements that have to be overtaken in the optimized skip list to place the extracted element to its new position

is comparable with the number of elements' swaps that are performed in DecreaseKey() operation in case of binary heap.

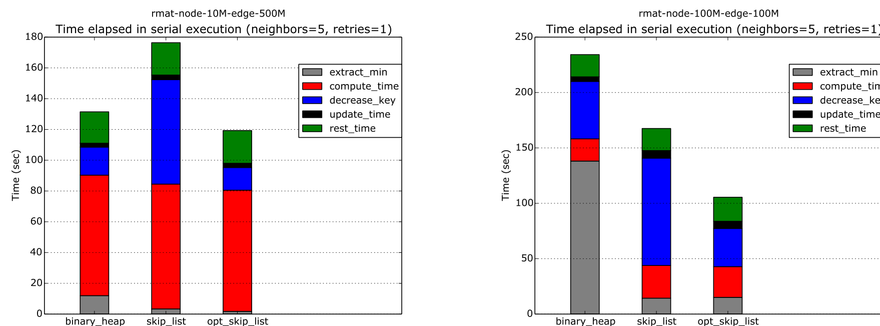


Figure 4.24: Time elapsed in serial execution for the three different structures used for the priority queue.

Subsequently, we evaluated the parallel execution of the algorithm for different number of threads. Figure 4.25 depicts the total runtime elapsed for the random-node-10M-edge-500M and the rmat-node-100M-edge-100M graph. As in the serial execution, the DecreaseKey() operation of the random-node-10M-edge-500M graph in case of using the simple skip list is much time-consuming and as a result the total runtime is the worst among the three executions. This is because the random-node-10M-edge-500M graph is very dense. The more dense a graph is, the more relaxations (DecreaseKey() operations) are performed due to the large number of edges. Thus, the DecreaseKey() operation is executed many times and the total execution time is increased significantly. On the contrary, the rmat-node-100M-edge-100M graph is a sparse graph and the DecreaseKey() is not executed so many times, since there are not many edges to cause relaxations, and the total runtime is not influenced. As we noticed in the serial execution, in this graph the ExtractMin() operation constitutes a great portion of the algorithm when using a binary heap for the priority queue. In this execution, the ExtractMin() operation depends on the number of vertices of the graph ($\mathcal{O}(\log n)$), while the skip list performs the ExtractMin() operation in a constant time ($\mathcal{O}(1)$). As a consequence, in the rmat-node-100M-edge-100M graph the binary heap execution has the worst total runtime.

Concerning the scalability of the parallel execution we observe that the execution with the optimized skip list achieves the highest scalability for both the rmat-node-10M-edge-500M graph and the rmat-node-100M-edge-100M graph. However, the rmat-node-100M-edge-100M graph does not appear good scalability, since this is a sparse graph and helper threads cannot offload many operations from the main thread. The skip list scales better because helper threads perform more relaxations and the main thread can offload more work. As we can see in figures 4.26a and 4.26b, the number of main thread's relaxations decreases much more in case of using a skip list. On the other hand, the execution with the binary heap cannot offload so much work. We suppose that the aborts performed when using the binary heap are useful aborts that would perform useful relaxations. Moreover,

the optimized skip list case scales better than the simple skip list case, as it needs a smaller read/write set and in this way, it avoids costly traversals and capacity transactional aborts. Lastly, in case of 28 threads, we can conclude that the NUMA effect degrades the scalability of all our executions due to expensive cache line transfers from one socket to another. The more cache lines are transferred from one socket to another, the worse performance the execution has.

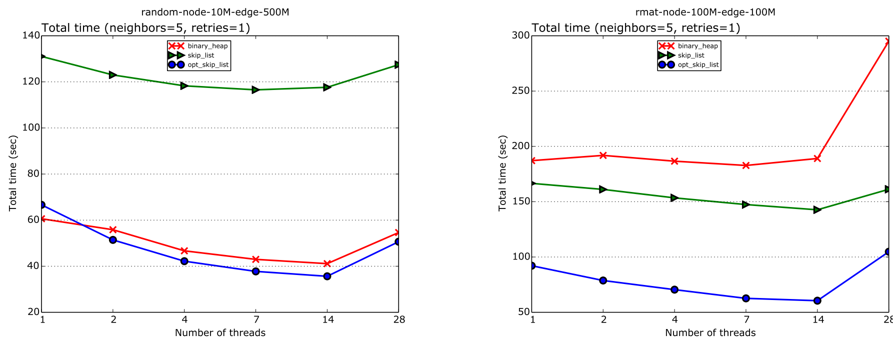
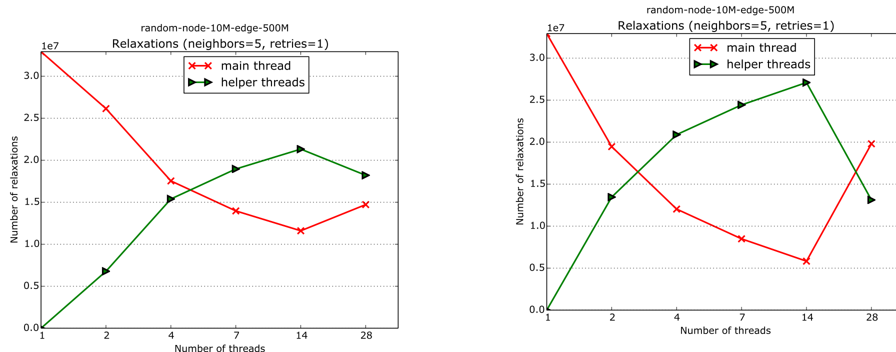


Figure 4.25: Time elapsed in parallel execution for the three different structures used for the priority queue.



(a) Using the binary heap structure.

(b) Using the optimized skip list structure.

Figure 4.26: Distribution of relaxations between the main and helper threads for the random-node-10M-edge-500M graph using two different structures for the priority queue.

Finally, we present the number of transactional commits/aborts of the main and helper threads in the three previous executions. Figure 4.27 depicts the number of commits/aborts of the main thread. The number of transactional aborts in the simple skip list execution is extremely high in comparison with the other two executions, especially in the larger graph. This is due to the large memory that the traversal of the DecreaseKey() operation

demands. The traversal in the simple skip list demands a very large read set and this may lead to transactional capacity aborts. Furthermore, there is also a higher probability of data transactional abort. The larger transactional sets that threads need, the higher probability of performing conflicting accesses among them it exists. Secondly, we can notice that the binary heap execution and the optimized skip list execution have comparable number of transactional aborts. In both of these executions the main thread suffers a really low number of transactional aborts and this means that helper threads do not obstruct main thread's progress. Furthermore, the number of transactional aborts is not importantly affected by the number of threads. It remains approximately the same as the number of threads increases, and we conclude that if the NUMA effect did not exist, we could have a better scalability.

As mentioned in previous section the NUMA effect influences the number of transactional aborts. Since the cache line transfers are expensive, a single transaction lasts a lot of time. As a consequence, data conflicts are more possible to be detected in longer transactions and there is a stronger probability of transactional abort due to time interrupt. When a transaction lasts more time than the time quantum, the scheduler of the operating system schedules out the process and the transaction is aborted. In the large rmat-node-100M-edge-100M graph, where many transactional cache lines have to be transferred in DecreaseKey() operation, the transaction is much time-consuming and this results to a large number of transactional aborts.

Similarly to the main thread, the number of transactional aborts of helper threads is increased in the simple skip list execution, as shown in figure 4.28. The traversal in the DecreaseKey() operation is very costly and causes many conflicts for helper threads, too. In the other two executions (binary heap and optimized skip list), the number of transactional aborts is relatively small and shows that the most of the concurrent accesses to the shared data structures are non-conflicting. Moreover, the number of transactional commits increases when adding more threads, since more threads perform relaxations. In this figure, the number of transactional commits in case of binary heap is higher than in the other two executions that use a skip list. Although the number of commits in case of binary heap is larger, these transactional commits do not result to useful relaxations. As we see in figures 4.26a and 4.26b, the optimized skip list structure results to more useful relaxations. We suppose that in case of using the binary heap structure, helper threads have more time to run until the main thread stops them. Thus, they may execute more than once the outer while loop (line 1 in listing 4.3) during the execution of one iteration (outer while loop) of the main thread. However, since helper threads do not extract elements from the priority queue, they will always read the same element in the ReadMin() function (the n-th helper thread reads always the n-th first element in the priority queue). Only when the main thread proceeds to its next iteration, helper threads will read another element to examine. Therefore, in case that helper threads have much time to run and they repeat more than once the outer while loop, they will perform the same process (same relaxations) more than once, executing relaxations for the one and only element that they can read in each main thread's iteration. They will perform many transactional commits (for the same element) without performing new useful relaxations.

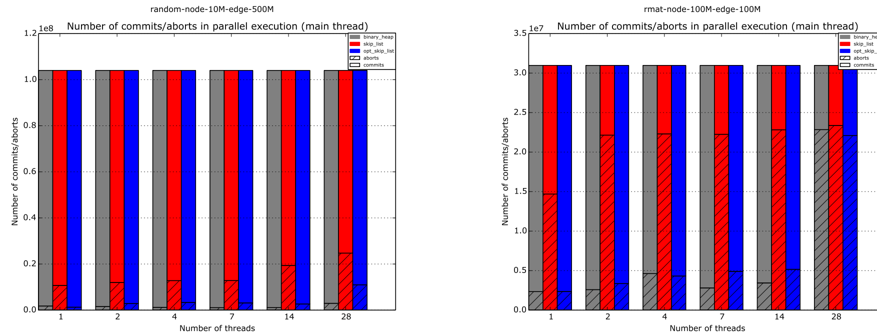


Figure 4.27: The number of commits/aborts of the main thread in parallel execution for the three different structures used for the priority queue.

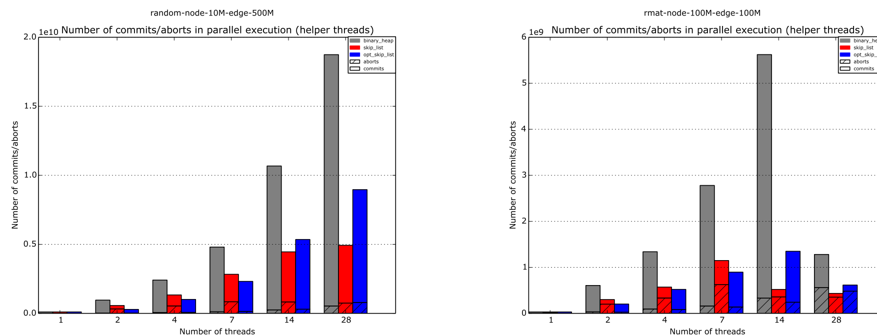


Figure 4.28: The number of commits/aborts of helper threads in parallel execution for the three different structures used for the priority queue.

4.7.3 Results

In the last part of our analysis we evaluated the performance of the parallel execution that employs our optimized skip list. We used the same graphs with the previous performance evaluation of the binary heap version. Figure 4.29 presents the speedups achieved using the optimized skip list in Dijkstra's algorithm. The proposed optimized scheme achieves significant speedups for all evaluated graphs. The maximum speedup achieved is 1.94 for the dense random-node-10M-edge-500M graph (14 threads).

As mentioned in the previous evaluation, the speedup is related to the density of the graph. For more dense graphs, the speedup is greater, as more parallelism can be exposed. Helper threads can offload more work from the main thread in dense graphs, where the number of edges is greater. Conversely, sparse graphs leave limited space for parallelism. In case of 28 threads the performance is degraded due to the NUMA effect.

Our implementation of the optimized skip list achieves higher performance speedup and scales better than the implementation with the binary heap. The main reason is the relaxations performed by the helper threads. As already mentioned, using the optimized

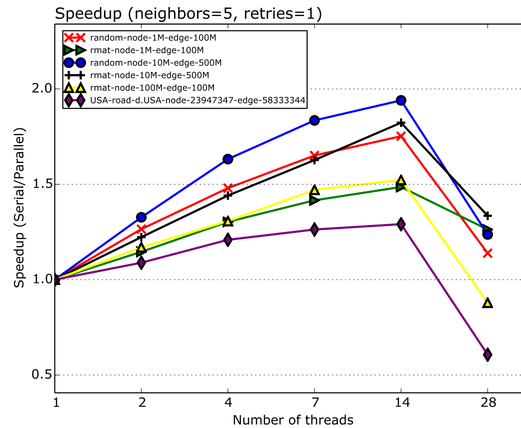


Figure 4.29: Multithreaded speedups for graphs of different density when using our optimized skip list.

skip list helper threads execute more useful relaxations. The number of relaxations performed using the optimized skip list is much larger than using the binary heap structure as figures 4.26a and 4.26b show, while these two implementations have comparable transactional aborts (figures 4.27 and 4.28). In some executions, the relaxations performed using our optimized skip list are double that performed when using the binary heap structure. This may be due to transactional aborts. We suppose that in case of binary heap the transactional aborts performed would result to useful relaxations. Since we cannot determine a policy when a conflict is detected, we are not sure for which transaction will be aborted. It is possible that a large number of the aborted transactions in case of using a binary heap would result to useful relaxations, while this does not happen when using our optimized skip list.

Secondly, the binary heap and the skip list are two completely different data structures. The skip list is a totally ordered data structure. Thus, at a given time, helper threads read the vertices with the first minimum distances (key) from the source (in ReadMin() function) to perform their relaxations. On the contrary, the bineary heap structure is not a totally ordered data structure. At a given time, helper threads read the elements that are located higher (in a small depth) in the binary heap, but these elements are not necessarily the vertices with the first minimum distances from the source. For example, in the binary heap of figure 4.30 if we had 5 helper threads, they would read the elements with the keys 9, 4, 26, 20 and 18. In case case of using a skip list, the 5 helper threads would examine the elements with the keys 4, 9, 11, 18 and 19, since skip list is totally ordered. Thus, we conclude that in each step of the algorithm helper threads examine different elements to perform their relaxations depending on the data structure used. We suppose that the total order that the skip list structure has, may lead helper threads to examine vertices that have obtained their optimal distance from the source with some stronger probability than that in case of using the binary heap structure and perform more useful relaxations than that

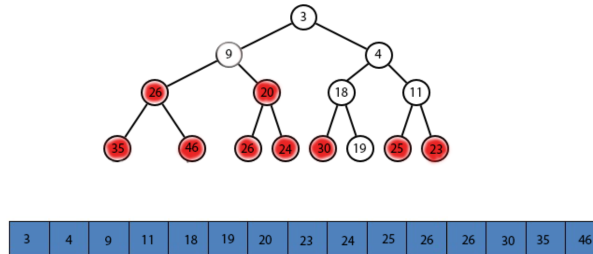


Figure 4.30: An example of binary heap. The vertices with the red color have not obtained their optimal distance from the source.

performed when using the binary heap structure.

Finally, to obtain an estimate of speedup we used the formula 4.6. It gives a speedup approximation based on main thread's relaxations. It also implies that the speedup should increase with the average out-degree. The more dense a graph is, the more parallelism can be exposed. However, this theoretical formula is a simple estimate and constitutes a theoretical upper bound for any performance improvement. It does not take into account the time spent in thread orchestration or delays due to consecutive transactional aborts. Figures 4.31 and 4.32 present a theoretical speedup and the speedup achieved for all evaluated graphs in case of 14 threads using the binary heap structure and the optimized skip list structure, respectively.

Graph	Ideal Speedup	Speedup achieved
random-node-1M-edge-100M	4	1.45
rmat-node-1M-edge-100M	3.58	1.27
random-node-10M-edge-500M	3.74	1.46
rmat-node-10M-edge-500M	3.09	1.35
rmat-node-100M-edge-100M	1.22	1.1
USA-road-node-23M-edge-58M	1.91	1.08

Figure 4.31: A speedup approximation based on main thread's relaxations in case of 14 threads and the speedup achieved using the binary heap structure.

Graph	Ideal Speedup	Speedup achieved
random-node-1M-edge-100M	4.57	1.75
rmat-node-1M-edge-100M	5.11	1.49
random-node-10M-edge-500M	5.14	1.94
rmat-node-10M-edge-500M	4.74	1.82
rmat-node-100M-edge-100M	1.79	1.52
USA-road-node-23M-edge-58M	1.48	1.29

Figure 4.32: A speedup approximation based on main thread's relaxations in case of 14 threads and the speedup achieved using the optimized skip list structure.

Chapter 5

Conclusion and Future Work

In the first part of this thesis we studied concurrent data structures, in particular binary search trees. Search trees are one of the most frequently used in a wider range of applications and parallelizing them introduces many challenges.

The first implementations presented constituted a naive approach for search tree data structure. We implemented binary search trees, AVL trees and Red-Black trees using coarse-grained and fine-grained locking technique for synchronization between threads. Our results demonstrate that coarse-grained implementations do not scale as they do not provide parallelism (serial execution) and fine-grained implementations scale in large trees until a small number of threads. Red-Black tree implementation has the highest performance, since this is a height-balanced tree.

On the other hand, more complex implementations scale better. They have a better synchronization mechanism, smaller contention windows and some of them perform a helping strategy. Secondly, according to the results presented there is no inherent difference between lock-free and lock-based algorithms. The main goal of more complex implementations is to reduce the number and the granularity of locks such that the execution to be close to the asynchronous algorithm. The closer to the sequential algorithm an implementation is, the better it scales. Finally, the scalability of synchronization is mostly a property of hardware. Synchronization primitives are non-scalable on NUMA architectures due to expensive cache line transfers. Therefore, the amount of synchronization on concurrent data structures must be reduced in order to achieve scalability on NUMA architectures.

Apart from the implementations examined on this thesis, there are still many interesting implementations of concurrent search trees to be studied, each of them has its own set of characteristics and innovations. For example, papers [25], [26], [27], [28] and [29] also present complex concurrent search trees to be studied. Furthermore, we could implement concurrent search trees using transactional memory as synchronization mechanism. Even a simple coarse-grained locking HTM implementation of a balanced tree like Red-Black tree could outperform lock-based and wait-free alternatives. However, to enable scalability to high numbers of threads, the programmer needs to be aware of the underlying HTM system's limitations and optimize the code appropriately.

Since the purpose of this analysis is to present concurrent data structures that scale efficiently and could be used in parallel software to improve its performance, besides search trees, there are many other data structures to be studied. Linked lists like FIFO queues, hash tables and priority queues like binary heaps and skip lists are also frequently used data structures and a performance and scalability analysis of such concurrent data structures would be quite challenging, too.

In the second part of this thesis, we applied some parallelization techniques to Dijkstra's algorithm, which is known to be hard to parallelize. We experimented on the algorithm proposed in [23], [24] in order to achieve high scalability in a real HTM system.

The aim of our experimentation is to find a policy that favors the main thread. Helper threads can perform operations simultaneously without interfering in main thread's progress. Thus, to avoid consecutive transactional aborts that would delay main thread's execution, we have forced main thread to acquire the global lock immediately. It can retry the execution of its work in transactional mode only once. Furthermore, helper threads can never acquire the global lock so as not to postpone main thread's execution and they always attempt to execute the critical section in transactional mode.

Secondly, we implemented a more coarse-grained transaction in main thread's code such that to reduce transactions' overhead and we tested the same for helper threads, too. However, performing a coarse-grained transaction can lead to capacity transactional aborts, especially in large graphs, due to the large part of the memory that this scheme accesses. We also experimented in padding technique used on the shared data structures. We concluded that padding does not affect the scalability of the algorithm, but it provides better runtime, as threads can exploit temporal and spatial locality.

Subsequently, we presented graphs for main thread's and helper threads' relaxations as well as the abort ratio of our executions. We can notice that the gain in main thread's relaxations is considerable, but there is no corresponding gain in speedup due to synchronization costs. Moreover, helper threads may perform relaxations that are not useful as the algorithms proceeds. Finally, our results also depict the effect of NUMA architecture. The performance collapses when threads are pinned in both two sockets of our system. As future work, it could be designed a variation of the presented algorithm in which the amount of synchronization would be reduced, and it would scale even in the presence of non-uniformity.

In the last part of this thesis we evaluated Dijkstra's algorithm employing a skip list instead of a binary heap for the priority queue. The results achieved using a simple skip list show that it has much worse performance than the binary heap version of the algorithm. The simple skip list requires a lot of memory and the traversal to its elements is very time-consuming. Thus, we implemented an optimized skip list that contains only discrete keys. Vertices that have the same key are stored to an internal nested list in skip list's element. In this way, our optimized skip list needs less memory, as there are many vertices with the same key during the execution of the algorithm and the traversal to its elements is much faster. This implementation achieves better performance than the binary heap and has higher scalability. Helper threads modify locally the optimized skip list and perform more useful relaxations than in case of using a binary heap.

Our optimized skip list does not contain one element for each vertex of the graph. The number of elements depends on the number of vertices that obtain the same distance (key in the skip list) from the source during the execution of the algorithm. It depends on the weights of graph's edges. As a consequence, we cannot conclude a fixed maximum height for the optimized skip list. We have to examine the average number of elements that our skip list has during the execution and set the maximum height as the logarithm of this average number. Generally, the parameter of the maximum height of our optimized skip list must be examined separately for each graph.

At a given step of the algorithm, the first element of our optimized skip list has the minimum distance-key from the source to be examined. This element may contain many ids of vertices of the graph which have the same minimum distance from the source at this step. Thus, a proposition would be that the main thread could extract the element from the skip list and delegate the different vertices (ids) to helper threads to perform their relaxations. In this way, there would be more than one settled node in a single step and this would result to greater gains. However, in case that a vertex is extracted from the priority queue, all of its edges have to be examined and all of its possible relaxations have to be executed. Otherwise, the algorithm will not be correct. Therefore, the algorithm have to be redesigned, such that each step (iteration of the outer loop) takes as much the most time consuming vertex examination among all threads (both the main and helper threads). In such an execution, the main thread should not stop helper threads when they still have edges to examine.

Our evaluation was performed in Intel's Haswell HTM. As continuation of the present work, the algorithm could be evaluated in other systems that support Hardware Transactional Memory. It could be explored the impact of various TM characteristics on the behavior of the presented schemes, such as resolution policy, version management and conflict detection. For example, an Hybrid conflict resolution policy which tends to favor older transactions against younger ones could be more efficient or a system with larger read/write transaction set would be more suitable for larger graphs. The programmer has to be informed of the underlying HTM system and its characteristics and change the code appropriately in order to achieve high scalability.

Eventually, in paper [24], results demonstrate interesting variations in the available parallelism between different execution phases. As the algorithm proceeds the available parallelism is reduced and the gains from the use of more helper threads are negligible. Thus, as future work it can be examined the different phases of the algorithm and explored more adaptive schemes in terms of the number of helper threads. Helper threads have to be dynamically adjusted as well as the tasks assigned to them.

Bibliography

[1] Amdahl, G., *The validity of the single processor approach to achieving large scale computing capabilities*, In Proceedings of AFIPS Spring Joint Computer Conference, Atlantic City, N.J., AFIPS Press, April 1967

[2] Flynn, M., *Some Computer Organizations and Their Effectiveness*, IEEE Transactions on Computers, 1972.

[3] Georgy Adelson-Velsky, G.; Evgenii Landis (1962)., *An algorithm for the organization of information*, Proceedings of the USSR Academy of Sciences (in Russian) 146: 263–266. English translation by Myron J. Ricci in Soviet Math. Doklady, 3:1259–1263, 1962.

[4] Leonidas J. Guibas and Robert Sedgewick (1978)., *A Dichromatic Framework for Balanced Trees*, Proceedings of the 19th Annual Symposium on Foundations of Computer Science. pp. 8–21. doi:10.1109/SFCS.1978.3.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein., *Introduction to Algorithms*, The MIT Press, 3rd ed., 2009.

[6] Siakavaras D., Nikas K., Goumas G., and Koziris N., *Performance analysis of concurrent red-black trees on htm platforms*, TRANSACT, 2015.

[7] Siakavaras D., Nikas K., Goumas G., and Koziris N., *Massively Concurrent Red-Black Trees with Hardware Transactional Memory*, 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), 2016.

[8] R. A. Tarjan., *Efficient top-down updating of red-black trees*, Tech. Rep. TR-006-85, Department of Computer Science, Princeton University, 1985.

[9] R. Bayer and M. Schkolnick., *Readings in database systems*, ch. Concurrency of Operations on B-trees, pp. 129–139, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1988.

[10] Tudor, D., Guerraoui, R., Trigonakis, V., *Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures*, In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, Pages 631-644. ASPLOS 2015.

[11] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun., *A Practical Concurrent Binary Search Tree*, PPoPP 2010.

[12] Drachsler D., Vechev, M.T., Yahav, E., *Practical concurrent binary search trees via logical ordering*, In: PPoPP, pp. 343-356. ACM(2014).

[13] Aravind Natarajan and Neeraj Mittal., *Fast Concurrent Lock-free*, Binary Search Trees. PPoPP 2014.

[14] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel., *Non-blocking Binary Search Trees*, PODC 2010.

[15] Maurice Herlihy., Nir Shavit., *The Art of Multiprocessor Programming*, Morgan Kaufmann Publishers Inc. San Francisco, CA, USA ©2008.

[16] Damron P., Fedorova A., Lev Y., Luchangco V., Moiv M., Nussbaum D., *Hybrid transactional memory*, In: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems, Pages 336-346. ASPLOS 2006.

[17] M. Moir., *Hybrid transactional memory*, July 2005.

[18] Casper J., Oguntebi T., Hong S., Bronson N., Kozyrakis C., Olukotun k., *Hardware acceleration of transactional memory on commodity systems*, In: Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems, Pages 27-38. ASPLOS 2011.

[19] A. Shriraman, M. F. Spear, H. Hossain, V. J. Marathe, S. Dwarkadas, and M. L. Scott., *An integrated hardware-software approach to flexible transactional memory*, SIGARCH Computer Architecture News, 35, June 2007.

[20] J. R. Larus and R. Rajwar., *Transactional Memory. Synthesis Lectures on Computer Architecture.*, Morgan & Claypool, 2007.

[21] Dijkstra, E. W., *A note on two problems in connection with graphs.*, Numerische Mathematik 1: 269–271. doi:10.1007/BF01386390, 1959.

[22] R. C. Prim, *Shortest connection networks and some generalizations.*, In: Bell System Technical Journal, 36 (1957), pp. 1389–1401.

[23] N. Anastopoulos, K. Nikas, G. Goumas, and N. Koziris, *Early experiences on accelerating dijkstra's algorithm using transactional memory.*, in Proc. 3rd Workshop on Multithreaded Architectures and Applications (MTAAP'09), 2009.

[24] K. Nikas, N. Anastopoulos, G. Goumas, N. Koziris, *Employing transactional memory and helper threads to speedup Dijkstra's algorithm.*, Parallel Processing, 2009. ICPP'09. International Conference on, 388-395.

[25] Crain, T., Gramoli, V., Raynal, M., *A speculation-friendly binary search tree*, In: PPoPP '12, Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming Pages 161-170, ACM New York, 2012.

[26] Chatterjee, B., Nguyen, N., Tsigas, P., *Efficient lock-free binary search trees*, In: PODC '14, Proceedings of the 2014 ACM symposium on Principles of distributed computing Pages 322-331, ACM New York, 2014.

[27] Prokopec, A., Bronson, N., Bagwell, P., Odersky, M., *Concurrent tries with efficient non-blocking snapshots*, In: PPoPP '12, Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming Pages 151-160 , ACM New York, 2012.

[28] Crain, T., Gramoli, V., Raynal, M., *A contention-friendly binary search tree*, In: Euro-Par'13, Proceedings of the 19th international conference on Parallel Processing Pages 229-240, Springer-Verlag Berlin, Heidelberg, 2013.

[29] Natarajan, A., Savoie, L., Mittal, N., *Concurrent Wait-Free Red Black Trees*, In: SSS 2013 Proceeding of the 15th International Symposium on Stabilization, Safety, and Security of Distributed Systems - Volume 8255 Pages 45-60, Springer-Verlag New York, 2013