ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Μελέτη και Σχεδίαση Παράλληλων Ουρών Προτεραιότητας σε Αρχιτεκτονικές Ανομοιόμορφης Πρόσβασης Μνήμης (NUMA)

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Φωτεινή Στρατή

Επιβλέπων: Γεώργιος Γκούμας
Επίκουρος Καθηγητής Ε.Μ.Π.

Αθήνα, Δεκέμβριος 2018

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Μελέτη και Σχεδίαση Παράλληλων Ουρών Προτεραιότητας σε Αρχιτεκτονικές Ανομοιόμορφης Πρόσβασης Μνήμης (NUMA)

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Φωτεινή Στρατή

Επιβλέπων: Γεώργιος Γκούμας
Επίκουρος Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 13η Δεκεμβρίου 2018 .

.....	.....	.....
Γ.Γκούμας	Ν.Κοζύρης	Ν.Παπασπύρου
Επίκουρος Καθηγητής Ε.Μ.Π	Καθηγητής Ε.Μ.Π	Αναπληρωτής Καθηγητής Ε.Μ.Π

Αθήνα, Δεκέμβριος 2018.

.....
Φωτεινή Στρατή
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Στις μέρες μας, τα πολυπήρυνα συστήματα αποτελούνται κυρίως από αρχιτεκτονικές ανομοιόμορφης πρόσβασης μνήμης (NUMA). Η υψηλή απόδοση και η κλιμακωσιμότητα είναι ζητούμενα των σύγχρονων εφαρμογών, οι οποίες βασίζονται σε παράλληλους αλγορίθμους δομών δεδομένων. Ο σχεδιασμός των αλγορίθμων αυτών θα πρέπει να γίνεται με γνώμονα τα χαρακτηριστικά των σύγχρονων αρχιτεκτονικών. Αντικείμενο της παρούσας διπλωματικής είναι οι παράλληλες δομές δεδομένων, και κυρίως οι ουρές προτεραιότητας. Μελετάμε και αξιολογούμε σε αρχιτεκτονικές ανομοιόμοφης πρόσβασης μνήμης έναν μεγάλο αριθμό παράλληλων υλοποιήσεων, κάποιες απο τις οποίες είναι ειδικά σχεδιασμένες για numa συστήματα (numa-aware υλοποιήσεις). Προτείνουμε μία νέα numa-aware ουρά προτεραιότητας (αλγόριθμος NUDDLE), η οποία αξιοποιεί τα χαρακτηριστικά των αρχιτεκτονικών τύπου NUMA και οδηγεί σε καλύτερα αποτελέσματα όταν ο ανταγωνισμός μεταξύ των νημάτων είναι έντονος. Παρόλα αυτά, σε εφαρμογές που χαρακτηρίζονται από υψηλά επίπεδα παραλληλισμού, οι non-numa-aware υλοποιήσεις επιτυγχάνουν μεγαλύτερη απόδοση. Συμπεραίνουμε, λοιπόν, ότι δεν υπάρχει μια ιδανική δομή δεδομένων και οτι οι συνθήκες στις οποίες εκτελείται μια εφαρμογή είναι αυτές που καθορίζουν το είδος του αλγορίθμου που πρέπει να χρησιμοποιηθεί. Στην προσπάθειά μας να προσεγγίσουμε τη λειτουργία μιας ιδανικής δομής, μοντελοποιούμε το πρόβλημα της επιλογής του καλύτερου αλγορίθμου και χρησιμοποιούμε τεχνικές μηχανικής μάθησης για να το επιλύσουμε. Πιο συγκεκριμένα, θεωρούμε δύο κλάσεις υλοποιήσεων (numa-aware και non-numa-aware) και προτείνουμε ένα μοντέλο βασισμένο σε δέντρα αποφάσεων, το οποίο αφού εκπαιδευτεί σε ένα ευρύ φάσμα πειραμάτων, κατηγοροιοποιεί τα διάφορα περιβάλλοντα εκτέλεσης στις δύο κλάσσεις με αρκετά υψηλή ακρίβεια. Στη συνέχεια, χρησιμοποιούμε αυτό το μοντέλο για να δημιουργήσουμε μια δυναμική δομή δεδομένων, η οποία, εξετάζοντας τις συνθήκες κάτω από τις οποίες εκτελείται, μπορεί να μεταβεί εύκολα και γρήγορα από numa-aware σε non-numa-aware αλγόριθμο επιτυγχάνοντας έτσι τη βέλτιστη δυνατή απόδοση. Η μεθοδολογία αυτή, πιστεύουμε ότι μπορεί να επεκταθεί και να χρησιμοποιηθεί για το σχεδιασμό και άλλων παράλληλων δομών δεδομένων, οι οποίες θα μπορούν να προσαρμόζονται στο εκάστοτε περιβάλλον, να αξιοποιούν τα χαρακτηριστικά της κάθε αρχιτεκτονικής και άρα να παρουσιάζουν τα βέλτιστα δυνατά αποτελέσματα.

Λέξεις-Κλειδιά: παράλληλες δομές δεδομένων, κλιμακωσιμότητα, ουρές προτεραιότητας, ανομοιόμορφη πρόσβαση μνήμης, numa-aware, μηχανική μάθηση, δέντρα αποφάσεων

Abstract

Nowadays, modern multicore processors are part of every computer system, and they are based on Non-Uniform Memory Access (NUMA) architectures. In order to fully leverage these machines, researchers should design efficient and scalable concurrent data structures that are aware of NUMA performance artifacts. In this thesis, we study both NUMA-oblivious and NUMA-aware concurrent data structures, particularly focusing on priority queues. We evaluate the scalability of the most known state-of-the-art concurrent priority queues on NUMA architectures, and propose a NUMA-aware concurrent priority queue, named NUDDLE, that outperforms state-of-the-art algorithms when contention between threads is high. However, in highly parallel benchmark configurations, NUDDLE performs poorly compared to numa-oblivious algorithms. Our experiments point that there is not an one-size-fits-all solution, since the profile of the application determines whether a numa-aware or a numa-oblivious implementation is more efficient. Towards that direction, we propose a self-aware concurrent priority queue. We formulate NUMA awareness as a classification problem and employ machine learning techniques to predict the more performant algorithm between a NUMA-aware and a NUMA-oblivious concurrent priority queue. We create a highly accurate profile-guided Decision Tree Classifier, trained on various microbenchmarks, that predicts the most appropriate algorithm taking as input the configurations of the execution. This classifier is utilized to create an adaptive priority queue, that according to its profile configurations can switch from numa-aware to non numa-aware implementation and vice versa when needed and achieve the highest available performance for all configuration workloads. We believe our methodology constitutes a general direction towards the design of dynamically adaptive concurrent data structures that perform best under all circumstances.

Keywords: concurrent data structures, scalability, priority queues, NUMA, numa-aware, numa-oblivious, classification, decision trees, self-aware

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 Επίκουρου Καθηγητή Ε.Μ.Π. Γεώργιου Γκούμα.

Θα ήθελα να ευχαριστήσω τους καθηγητές μου κ. Γκούμα, κ. Κοζύρη και κ. Παπασπύρου για τη διδασκαλία τους και τις γνώσεις που μου προσέφεραν όλα αυτά τα χρόνια, καθώς αποτέλεσαν την έμπνευσή μου για να ασχοληθώ εκτενώς με την επιστήμη των υπολογιστών.

Ιδιαιτέρως θα ήθελα να ευχαριστήσω την υποψήφια διδάκτωρ Χριστίνα Γιαννούλα για τις γνώσεις της, τη βοήθεια, την αμέριστη στήριξη, την ενθάρρυνση και την καθοδήγηση που μου προσέφερε καθώς και για την καλή της διάθεση και το χρόνο που μου αφιέρωσε. Επίσης θα ήθελα να ευχαριστήσω τον υποψήφιο διδάκτωρ Δημήτριο Σιακαβάρα για την καθοδήγησή του, τις συμβουλές και τις τεχνικές του γνώσεις.

Τέλος, θα ήθελα να ευχαριστήσω την οικογένεια και τους φίλους μου για την αγάπη, τη στήριξη και την εμπιστοσύνη που μου δείχνουν όλα αυτά τα χρόνια.

Φωτεινή Στρατή,
Δεκέμβριος 2018

Contents

List of Figures

Chapter 1

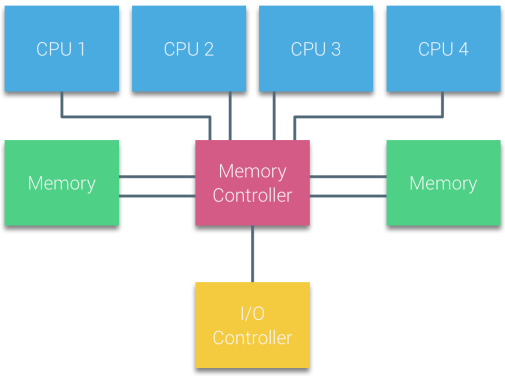
Εκτεταμένη Περίληψη

Στο παρόν κεφάλαιο παρουσιάζουμε μια εκτεταμένη περίληψη της διπλωματικής εργασίας. Αρχικά, θα αναφέρουμε τα βασικά χαρακτηριστικά των αρχιτεκτονικών ανομοιόμορφης πρόσβασης μνήμης (NUMA) και τις βασικές αρχές του παράλληλου προγραμματισμού. Στη συνέχεια θα εμβαθύνουμε στις τεχνικές που χρησιμοποιήσαμε, θα παρουσιάσουμε τη διαδικασία που ακολουθήσαμε, τα αποτελέσματα των πειραμάτων μας καθώς και μελλοντικές επεκτάσεις.

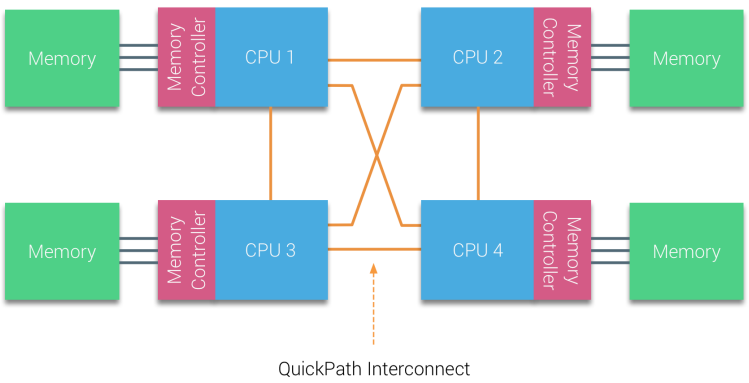
1.1 Αρχιτεκτονικές Ανομοιόμορφης Πρόσβασης Μνήμης (NUMA)

Στην εικόνα 1.1 παρουσιάζεται μια απλοποιημένη αναπαράσταση ενός συστήματος αρχιτεκτονικής ομοιόμορφης πρόσβασης μνήμης (UMA). Όλοι οι επεξεργαστές μπορούν να προσπελάσουν ομοιόμορφα (στον ίδιο χρόνο) οποιοδήποτε τμήμα της κύριας μνήμης χρησιμοποιώντας το δίαυλο μεταξύ μνήμης και κεντρικής μονάδας επεξεργασίας (cpu). Η αρχιτεκτονική αυτή χρησιμοποιείται σε συμμετρικά πολυπήρυνα μηχανήματα (SMP). Τα μηχανήματα αυτά δεν μπορούν να κλιμακώσουν επαρκώς, καθώς περιορίζονται από το εύρος ζώνης του διαύλου επικοινωνίας μνήμης-cpu. Για το σκοπό αυτό, τα σύγχρονα συστήματα βασίζονται κυρίως σε αρχιτεκτονικές ανομοιόμορφης πρόσβασης μνήμης (NUMA).

Σε ένα σύστημα NUMA, ο συνολικός αριθμός επεξεργαστών χωρίζεται σε ομάδες-κόμβους (nodes), οι οποίοι συνδέονται μέσω ενός αποδοτικού δικτύου διασύνδεσης. Ο κάθε κόμβος διαθέτει ένα τμήμα της συνολικής μνήμης καθώς και ξεχωριστή μονάδα ελέγχου μνήμης. Η εικόνα 1.2 αναπαριστά ένα σύστημα με τέσσερις κόμβους NUMA (NUMA nodes). Το τμήμα της μνήμης που βρίσκεται πλησίον ενός κόμβου ορίζεται ως τοπική μνήμη για το συγκεκριμένο κόμβο (καθώς και και για τους επεξεργαστές που τον αποτελούν), ενώ τα υπόλοιπα τμήματα αναφέρονται ως απομακρυσμένη μνήμη. Ένας επεξαργαστής μπορεί να προσπελάσει την τοπική του μνήμη πολύ πιο γρήγορα και αποδοτικά από ότι την απομακρυσμένη, η οποία προϋποθέτει τη διάσχιση του δικτύου διασύνδεσης.



Σχήμα 1.1: Αρχιτεκτονική Ομοιόμορφης Πρόσβασης Μνήμης



Σχήμα 1.2: Αρχιτεκτονική Ανομοιόμορφης Πρόσβασης Μνήμης

Όταν σε ένα σύστημα τύπου NUMA διατηρείται η συνοχή της κρυφής μνήμης (cache coherence), το σύστημα αναφέρεται ως ccNUMA. Στις μέρες μας, οι εφαρμογές για non-cache-coherent NUMA συστήματα είναι ελάχιστες, για αυτό και τα περισσότερα συστήματα ανομοιόμορφης πρόσβασης μνήμης χαρακτηρίζονται ως ccNUMA. Προκειμένου να διατηρηθεί η συνοχή της κρυφής μνήμης και να εξυπηρετηθούν τα αιτήματα των επεξεργαστών στη μνήμη, κάθε κόμβος είναι ενήμερος για τις θέσεις των διάφορων τμημάτων μνήμης στο σύστημα. Όταν ένας επεξεργαστής στέλνει ένα αίτημα μνήμης, εξετάζονται αρχικά τα επίπεδα ιεραρχίας της κρυφής μνήμης (L1, L2, L3), στη συνέχεια η τοπική στον κόμβο μνήμη και τέλος η απομακρυσμένη μνήμη. Όταν το σχετικό κομμάτι μνήμης βρίσκεται σε απομακρυσμένο κόμβο, τα δεδομένα πρέπει να μεταφερθούν μέσω του δικτύου στην κρυφή μνήμη του αιτούντος επεξεργαστή. Η διαδικασία αυτή γίνεται αυτόματα σε ένα NUMA σύστημα, σε αντίθεση με τα συστήματα κατανεμημένης μνήμης, στα οποία ο προγραμματιστής είναι υπεύθυνος για την επικοινωνία μεταξύ των επεξεργαστών και την ανταλλαγή δεδομένων μεταξύ των κόμβων.

Το μέγιστο εύρος ζώνης ενός συστήματος NUMA ορίζεται ως το άθροισμα του εύρους

ζώνης κάθε ελεγκτή μνήμης. Στα σύγχρονα συστήματα NUMA ανταλάσσονται μηνύματα διατήρησης συνοχής της κρυφής μνήμης μόνο όταν αυτό απαιτείται, και συνεπώς το μέγιστο εύρος ζώνης επιτυγχάνεται όταν όλοι οι επεξεργαστές προσπελαύνουν την τοπική τους μνήμη. Έτσι, μειώνεται ο αριθμός των μηνυμάτων που αποστέλλονται μεταξύ των κόμβων, και άρα μειώνεται η κίνηση στο δίκτυο διασύνδεσης. Το γεγονός αυτό έχει ως αποτέλεσμα την αύξηση του εύρους ζώνης και άρα της απόδοσης του συστήματος.

1.2 Βασικές Αρχές του Παράλληλου Προγραμματισμού

1.2.1 Ο Νόμος του Amdahl

Μια εφαρμογή που εκτελείται σε ένα μηχάνημα με N πυρήνες θα έπρεπε, ιδανικά, να παρουσιάζει N φορές καλύτερη απόδοση σε σχέση με ένα μονοπύρηνο σύστημα. Στην πραγματικότητα, κάτι τέτοιο δε συμβαίνει, καθώς το κόστος της επικοινωνίας και του συγχρονισμού μεταξύ των πολλαπλων επεξεργαστών δεν μπορεί να εξαληφθεί. Το φαινόμενο αυτό περιγράφεται από το Νόμο του Amdahl: Ο βαθμός στον οποίο μπορεί να κλιμακώσει μια σύνθετη εργασία εξαρτάται από το τμήμα αυτής που πρέπει να εκτελεστεί σειριακά. Πιο συγκεκριμένα, η επιτάχυνση S μιας παράλληλης εργασίας ορίζεται ως ο χρόνος εκτέλεσης της εργασίας σε έναν επεξεργαστή προς το χρόνο ο οποίος χρειάζεται για την παράλληλη εκτέλεση αυτής της εργασίας σε ένα σύστημα N επεξεργαστών. Επιπλέον, ας συμβολίσουμε με p το τμήμα της εργασίας που μπορεί να εκτελεστεί παράλληλα και ας θεωρήσουμε ότι ο χρόνος εκτέλεσής της σε έναν επεξεργαστή είναι 1. Τότε, το σειριακό τμήμα της εργασίας είναι 1-p και ο χρόνος που απαιτείται για την εκτέλεση σε N επεξεργαστές ισούται με p/N. Συνεπώς για την εκτέλεση σε ένα N-πύρηνο σύστημα ο συνολικός χρόνος είναι:

$$T = 1 - p + \frac{p}{N}$$

Σύμφωνα με τον Amdahl, η επιτάχυνση, δηλαδή ο λόγος ανάμεσα στον χρόνο εκτέλεσης σε ένα μονοπύρηνο και σε ένα πολυπήρυνο σύστημα δίνεται από τον τύπο:

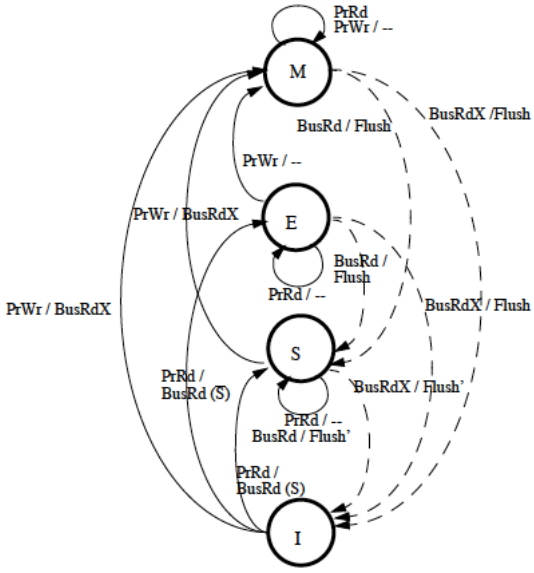
$$S = \frac{1}{1 - p + \frac{p}{N}}$$

Συμπεραίνουμε λοιπόν ότι όταν $N \to \infty, S \to \frac{1}{1-p}$.

Ο Νόμος του Amdahl δηλώνει ότι η επιτάχυνση οποιασδήποτε παράλληλης εργασίας περιορίζεται από το τμήμα του προγράμματος το οποίο πρέπει να εκτελεστεί σειριακά και οφείλεται σε εξαρτήσεις μεταξύ των δεδομένων και στο κόστος συγχρονισμού μεταξύ των πολλαπλών νημάτων εκτέλεσης. Συνεπώς, για να αυξήσουμε την κλιμακωσιμότητα μιας παράλληλης εφαρμογής, θα πρέπει να παραλληλοποιήσουμε όσο το δυνατόν μεγαλύτερο τμήμα της.

1.2.2 Συνοχή Μνήμης (Memory Coherence)

Οι κρυφές μνήμες (cache) αποθηκεύουν αντίγραφα δεδομένων της κύριας μνήμης που έχουν χρησιμοποιηθεί πρόσφατα από τον εκάστοτε επεξεργαστή. Συνεπώς, είναι πολύ πιο εύκολο και γρήγορο για ένα νήμα εκτέλεσης να προσπελάσει δεδομένα που βρίσκονται στην τοπική του κρυφή μνήμη, γι'αυτό και οι σύγχρονες αρχιτεκτονικές βασίζονται σε ιεραρχίες κρυφής μνήμης. Παρόλο που οι κρυφές μνήμες μειώνουν το μέσο χρόνο πρόσβασης στα δεδομένα και την κίνηση στο δίκτυο διασύνδεσης, η ύπαρξη πολλαπλών αντιγράφων της ίδιας περιοχής κύριας μνήμης δημιουργεί προβλήματα σε πολυπήρηνα συστήματα, στα οποία δύο ή περισσότερα νήματα προσπελαύνουν ταυτόχρονα διαφορετικά αντίγραφα της μνήμης. Εάν κανένα από αυτά τα νήματα δεν επιχειρήσει να μεταβάλει κάποιο αντίγραφο, τότε τα δεδομένα μπορούν χωρίς πρόβλημα να μοιράζονται μεταξύ των νημάτων. Εάν όμως συμβεί κάποια αλλαγή σε ένα αντίγραφο, τότε τα υπόλοιπα αντίγραφα δε θα είναι έγκυρα και το σύστημα δε θα θεωρείται πλέον συνεπές. Για το σκοπό αυτο, είναι απαραίτητος ένας μηχανισμός ο οποίος θα ενημερώνει το σύστημα για αλλαγές σε κοινά δεδομένα: ο μηχανισμός αυτός είναι γνωστός και ως **πρωτόκολλο συνοχης κρυφής μνήμης**.



Σχήμα 1.3: MESI πρωτόκολλο

Τα πρωτόκολλα συνοχής μνήμης βασίζονται είτε σε τεχνικές ανίχνευσης στο δίαυλο επικοινωνίας ή στη διατηρηση καταλόγου δεδομένων. Ένα από τα πιο γνωστά πρωτόκολλα είναι το **MESI** το οποίο φαίνεται στην εικόνα 1.3.

Σύμφωνα με το MESI πρωτόκολλο, μια γραμμή κρυφής μνήμης (cache line) μπορεί να βρίσκεται σε μία από τις ακόλουθες καταστάσεις:

- **Modified-Τροποποιημένη:** Μόνο η συγκεκριμένη cache line περιέχει την πιο πρόσφατη εκδοχή των δεδομένων που δεν είναι ίδια με αυτά της κύρια μνήμης. Συνε-

πώς, θα πρέπει σύντομα το περιεχόμενο της cache line να αντιγραφεί στην κύρια μνήμη.

- **Exclusive-Αποκλειστική:** Μόνο η συγκεκριμένη cache line περιέχει την πιο πρόσφατη εκδοχή των δεδομένων τα οποία συμπίπτουν με αυτά της κύριας μνήμης και μπορεί να μεταβεί σε κατάσταση Modified μετά από εγγραφή στα δεδομένα από τον επεξεργαστή. Εναλλακτικά, η κατάσταση της cache line μπορεί να αλλάξει σε Shared έαν υπάρξει κάποιο αίτημα ανάγνωσης των δεδομένων από κάποιον άλλον επεξεργαστή.

- **Shared-Μοιραζούμενη:** Η cache line περιέχει την πιο πρόσφατη εκδοχή των δεδομένων. Η εκδοχή αυτή περιέχεται και στην κύρια μνήμη, πιθανόν και σε άλλες κρυφές μνήμες. Έπειτα από ένα αίτημα εγγραφής στα δεδομένα, η cache line μπορεί να μεταβεί είτε σε κατάσταση Modified (εάν το αίτημα προήλθε από τον ίδιο επεξεργαστή) είτε σε κατάσταση Invalid (εάν το αίτημα προήλθε από κάποιον άλλο επεξεργαστή).

- **Invalid-Άκυρη:** Τα δεδομένα που περιέχονται στην cache line δεν είναι έγκυρα (έχουν μεταβληθεί).

Επεκτάσεις του MESI είναι τα πρωτόκολλα MOESI και MESIF.

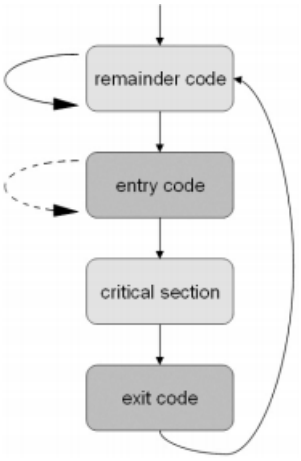
1.2.3 Μηχανισμοί Συγχρονισμού

Στα σύγχρονα πολυπήρυνα συστήματα, τα δεδομένα προσπελαύνονται ταυτόχρονα από πολλαπλά νήματα εκτέλεσης δημιουργώντας την ανάγκη για κάποιο μηχανισμό συγχρονισμού. Στόχος του εκάστοτε μηχανισμύ είναι να διασφαλίσει ότι τα κρίσιμα τμήματα ενός παράλληλου προγράμματος εκτελούνται κάθε φορά μόνο από ένα νήμα. Στη συνέχεια θα παρουσιάσουμε τρεις βασικές κατηγορίες μηχανισμών συγχρονισμού. Ο κατάλληλος μηχανισμός επιλέγεται ανάλογα με τις ανάγκες της εφαρμογής και τα χαρακτηριστικά της πλατφόρμας στην οποία εκτελείται.

- **Αμοιβαίος Αποκλεισμός (Mutual Exclusion):**

  Ο μηχανισμός αυτός χρησιμοποείται για το σχεδιασμό blocking υλοποιήσεων (υλοποιήσεις στις οποίες ένα ή περισσότερα νήματα βρίσκονται σε κατάσταση αναμονής πριν εισέλθουν στο κρίσιμο τμήμα). Οι υλοποιήσεις αυτές ακολουθούν το πρότυπο που παρουσιάζεται στην εικόνα 1.4. Απαιτείται πολύ προσεκτικός σχεδιασμός του κώδικα πριν το κρίσιμο τμήμα ώστε να αποφευχθούν φαινόμενα αδιεξόδου ή λιμοκτονίας διεργασιών.

  Αμοιβαίο Αποκλεισμό μπορούμε να έχουμε χρησιμοποιώντας σημαφόρους, mutexes και locks, δηλαδή μεταβλητές των οποίων οι τιμές επιτρέπουν ή όχι την πρόσβαση στο κρίσιμο τμήμα. Στις σύγχρονες εφαρμογές, ο αμοιβαίος αποκλεισμός επιτυγχάνεται είτε με coarse grained είτε με fine grained locking. Στον πρώτο τρόπο κλειδώματος υπάρχει συνήθως ένα κύριο lock για όλα τα δεδομένα, ενώ στο δεύτερο



Σχήμα 1.4: Μηχανισμός Αμοιβαίου Αποκλεισμού

τρόπο χρησιμοποιούνται περισσότερα locks και το καθένα προστατεύει λιγότερα δεδομένα. Παρόλο που η χρήση coarse grained locks οδηγεί σε απλούστερες υλοποιήσεις, το MESI πρωτόκολο μπορεί να έχει αρνητική επιδραση στην απόδοση και στην κλιμακωσιμότητα αυτών των υλοποιήσεων. Πολλαπλά νήματα ανταγωνίζονται συνεχώς για το ίδιο lock, άρα για την ίδια cache line, οδηγώντας σε συνεχείς παραβιάσεις της συνοχής μνήμης και άρα σε ενεργοποιήσεις του πρωτοκόλλου MESI, αυξάνοντας έτσι την κίνηση στο δίκτυο διασύνδεσης.

- **Ατομικές Λειτουργίες (Atomic Operations):**

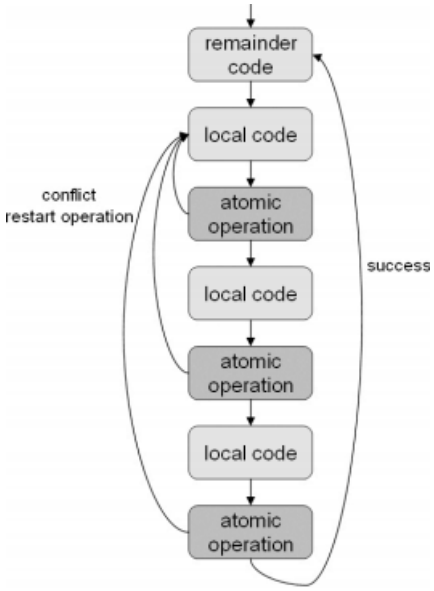
  Ο συγχρονισμός μεταξύ των νημάτων μπορεί να επιτευχθεί και με τη χρήση ατομικών λειτουργιών όπως φαίνεται και στην εικόνα 1.5. Οι αντίστοιχες υλοποιήσεις χαρακτηρίζονται και ως non-blocking: τα νήματα εκτελούν ατομικές λειτουργίες όταν θέλουν να μεταβάλλουν μοιραζόμενα δεδομένα και είτε επιτυγχάνουν είτε ξαναπροσπαθούν.

  Η ατομικότητα μιας λειτουργίας σημαίνει ότι πραγματοποιείται χωρίς παρεμβολές, και τα υπόλοιπα νήματα βλέπουν απλά το αποτέλεσμά της. Συνεπώς, μια ατομική λειτουργία είτε επιτυγχάνει είτε αποτυγχάνει στο σύνολό της. Παραδείγματα ατομικών εντολών είναι: atomic increment, atomic exchange register and memory location, και compare and swap (CAS).

  Οι ατομικές εντολές είναι πιο γρήγορες από τα locks, μπορούν να οδηγήσουν σε υψηλότερα επίπεδα παραλληλισμού, και χρησιμοποιούνται ευρέως στο σχεδιασμό αποδοτικών παράλληλων δομών δεδομένων.

- **Transactional Memory:**

  Ένας εναλλακτικός τρόπος σχεδιασμού non-blocking υλοποιήσεων είναι με χρήση transactional memory. Ως transaction (συναλλαγή) θεωρούμε ένα σύνολο εντολών που εκτελούνται ατομικά από ένα νήμα. Μια συναλλαγή μπορεί είτε να επιτύχει



Σχήμα 1.5: Υλοποίηση με ατομικές λειτουργίες

(commit) είτε να αποτύχει (abort). Εάν δεν υπάρχουν συγκρούσεις για κοινά δεδομένα την ίδια χρονική στιγμή, η συναλλαγή θα επιτύχει και θα πραγματοποιήσει τις εκάστοτε αλλαγές στα δεδομένα, οι οποίες πλέον θα είναι ορατές και από τα υπόλοιπα νήματα. Σε αντίθετη περίπτωση, η συναλλαγή αποτυγχάνει και δεν μεταβάλει τα δεδομένα. Οι βασικοί τύποι της συγκεκριμένης μεθόδου είναι Software Transactional Memory (STM) και Hardware Transactional Memory (HTM).

```
transaction_begin()
        ...
        ...
        critical section
        ...
        ...
transaction_end()
```

Σχήμα 1.6: Τα Transactional Blocks χρησιμοποιούνται για να προστατέψουν τα κρίσιμα τμήματα ενός προγράμματος.

Σε κάθε περίπτωση, τα υπόλοιπα νήματα θα αντιληφθουν το αποτέλεσμα μιας συναλλαγής και ποτέ μια ενδιάμεση κατάσταση. Όταν ένα τμήμα κώδικα πρέπει να εκτελεσθεί ατομικά, το μόνο που χρειάζεται είναι να δηλωθεί σε ένα transaction block όπως φαίνεται και στην εικόνα 1.6. Συνεπώς, η συγκεκριμένη μέθοδος συγχρονισμού προσφέρει ένα υψηλο επίπεδο αφαίρεσης και διευκολύνει πολύ τον προγραμματιστή στο σχεδιασμό παράλληλων υλοποιήσεων. Παρόλα αυτά το οποιοδήποτε σφάλμα είναι δύσκολο να ανιχνευτεί και να επιλυθεί.

1.3 Αξιολόγηση Παράλληλων Ουρών Προτεραιότητας σε NUMA Συστήματα

Στην παρούσα διπλωματική εργασία επικεντρωθήκαμε στις ουρές προτεραιότητας (priority queues). Πρόκειται στην ουσία για αφηρημένες δομές δεδομένων, οι οποίες ορίζουν τις πιθανές λειτουργίες στα στοιχεία τους χωρίς να τις υλοποιούν. Σε μια ουρά προτεραιότητας, κάθε στοιχείο συνδέεται με ένα συγκεκριμένο βαθμό ή αλλιώς μια προτεραιότητα. Η σχέση ανάμεσα στο βαθμό και στην προτεραίοτητα του εκάστοτε στοιχείου είναι αντίστροφη: μικρή τιμή βαθμού σημαίνει μεγάλη προτεραιότητα. Οι λειτουργίες αυτής της δομής είναι βασικά η εύρεση και η διαγραφή του στοιχείου με το μικρότερο βαθμό, δηλαδή με τη μεγαλύτερη προτεραιότητα (findMin() και deleteMin() αντίστοιχα), καθώς και η εισαγωγή (insert()) ενός νέου στοιχείου. Για να υλοποιηθούν οι συγκεκριμένες λειτουργίες χρησιμοποιείται σα βάση κάποια άλλη δομή δεδομένων, κυρίως δυαδικοί σωροί ή skip lists.

Οι ουρές προτεραιότητας χρησιμοποιούνται ευρέως σε αλγορίθμους γράφων όπως στους αλγορίθμους Dijkstra και Prim, καθώς και στον αλγόριθμο A*. Χρησιμοποιούνται επίσης σε περιπτώσεις διαχείρισης περιορισμένων πόρων όπως το εύρος ζώνης γραμμών μεταφοράς από ένα δρομολογητή δικτύου, για εξισσορόπηση φορτίου σε servers, για διαχείριση διακοπών και γενικά σε περιπτώσεις στις οποίες μπορεί να προκύψουν ξαφνικά γεγονότα υψηλής προτεραιότητας και να απαιτούν άμεση εξυπηρέτηση.

Λόγω, λοιπόν, της σημαντικότητας τους, έχουν προταθεί πολλές παράλληλες υλοποιήσεις ουρών προτεραιότητας. Οι συγκεκριμένες δομές, όμως, δεν είναι εύκολο να παραλληλοποιηθούν αποτελεσματικά. Το πρόβλημα έγκειται στη λειτουργία deleteMin() κατά την οποία τα νήματα ανταγωνίζονται για τα ίδια τμήματα μνήμης. Για το σκοπό αυτό, έχουν προταθεί υλοποιήσεις οι οποίες ονομάζονται ως ”χαλαρές ουρές προτεραιότητας” [2] και σύμφωνα με τις οποίες κατά τη deleteMin() τα νήματα πλέον επιθυμούν να διαγράψουν ένα από τα στοιχεία μικρότερης προτεραιότητας. Έτσι, οι υλοποιήσεις αυτές στοχεύουν στο να μειώσουν τον ανταγωνισμό μεταξύ των νημάτων.

Στην παρούσα διπλωματική μελετήσαμε τη συμπεριφορά παράλληλων ουρών προτεραιότητας σε ένα σύστημα NUMA. Παρατηρήσαμε, λοιπόν, ότι όταν η εκτέλεση περιοριζόταν σε ένα NUMA κόμβο, οι περισσότεροι αλγόριθμοι κλιμάκωναν επαρκώς. Όταν όμως ο αριθμός των νημάτων που επιχειρούσαν να προσπελάσουν τη δομή ήταν μεγαλύτερος από το διαθέσιμο αριθμό πυρήνων ενός NUMA κόμβου, η απόδοση των αλγορίθμων μειώνοταν δραματικά. Η βασική αιτία είναι όπως αναφέραμε η λειτουργία deleteMin() η οποία αυξάνει τον ανταγωνισμό μεταξύ των νημάτων, οδηγεί σε διαρκείς ανταλλαγές μηνυμάτων μεταξύ των κόμβων και αυξάνει την κίνηση στο δίκτυο.

1.4 Numa-aware Δομές Δεδομένων

Οι αρχιτεκτονικές ανομοιόμορφης μνήμης επιτρέπουν σε ένα σύστημα να κλιμακώσει σε έναν μεγάλο αριθμό πηρύνων. Όμως, το περιορισμένο εύρος ζώνης του δικτύου διασύνδεσης μπορεί να έχει δραματική επίδραση στην απόδοση ενός παράλληλου αλγο-

ρίθμου. Για να αξιοποιηθεί στο έπακρον ένα NUMA σύστημα, θα πρέπει οι γνωστοί μας αλγόριθμοι να επανασχεδιαστούν ώστε να γίνουν καταλληλότεροι για τις NUMA αρχιτεκτονικές. Στη συνέχεια της εργασίας θα αναφερόμαστε με τον όρο numa-aware σε ένας αλγόριθμο σχεδιασμένο με γνώμονα τα χαρακτηριστικά ενός συστήματος NUMA. Επειδή, όπως γνωρίζουμε, οι δομές δεδομένων αποτελούν βασικό στοιχείο των παράλληλων αλγορίθμων, στόχος είναι ο σχεδιασμός numa-aware δομών δεδομένων.

Όπως αναφέραμε, σε ένα NUMA σύστημα μια τοπική πρόσβαση μνήμης είναι πολύ πιο γρήγορη από μια απομακρυσμένη. Έτσι, ένας λανθασμένος σχεδιασμός που επιφέρει μεγάλο αριθμό απομακρυσμένων προσβάσεων στη μνήμη, αυξάνει την κίνηση στο δίκτυο και μειώνει την απόδοση της εφαρμογής. Επιπλέον, όταν υπάρχει έντονος ανταγωνισμός μεταξύ των νημάτων για τα ίδια κομμάτια μνήμης (όπως συμβαίνει στην περίπτωση των ουρών προτεραιότητας), υπάρχει και πάλι έντονη κίνηση στο δίκτυο λόγω του πρωτοκόλλου MESI. Το φαινόμενο κατά το οποίο η αυξημένη κίνηση στο δίκτυο μειώνει την απόδοση ενός αλγορίθμου σε ένα NUMA σύστημα, είναι γνωστό ως numa effect. Συνεπώς, το βασικό χαρακτηριστικό των numa-aware υλοποιήσεων είναι ότι επιδιώκουν να περιορίσουν όσο το δυνατό περισσότερο το numa effect και την επικοινωνία μεταξύ των NUMA κόμβων. Στη συνέχεια θα αναφερθούμε σε δύο βασικές black-box τεχνικές για το σχεδιασμό numa-aware δομών δεδομένων.

1.5 Οι τεχνικές Combining και Delegation

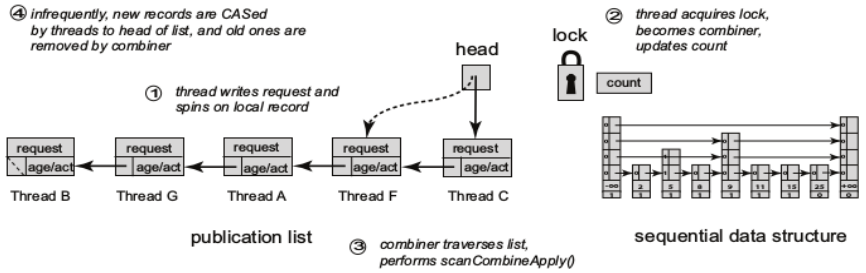
1.5.1 Λειτουργία δια αντιπροσώπου (Delegation)

Η τεχνική της εκτέλεσης λειτουργιών δια αντιπροσώπου (delegation) αποτελεί τη βάση πολλών numa-aware δομών δεδομένων. Σύμφωνα με αυτή την τεχνική, μόνο ένα νήμα, το οποίο θεωρείται ο server, έχει πρόσβαση στη δομή. Τα υπόλοιπα νήματα θεωρούνται clients, δεν μπορούν να προσπελασουν τη δομή και τα αιτήματά τους εξυπηρετούνται από το server: όταν κάποιος client θέλει να προσπελάσει τη δομή στέλνει ένα αίτημα στο server και περιμένει την απάντησή του. Ο server, δηλαδή, λαμβάνει αιτήματα από τους clients, εκτελεί τις αντίστοιχες λειτουργίες στη δομή και στέλνει τα αποτελέσματα των λειτουργιών αυτών πίσω στους clients.

Σε αρχιτεκτονικές κοινής μνήμης, οι servers και οι clients επικοινωνούν διαβάζοντας και γράφοντας στα ίδια τμήματα μνήμης προκαλώντας την ενεργοποίηση του πρωτοκόλλου MESI, κάτι που όπως είδαμε μπορεί επιδράσει αρνητικά στην απόδοση ενός αλγορίθμου. Η σχέση μεταξύ του delegation και των διαφόρων μηχανισμών διατήρησης συνοχής μνήμης έχει αποτελέσει αντικείμενο ερευνών και έχουν προταθεί πιθανοί τρόποι βελτίωσης και επέκτασης. Η τεχνική delegation μπορεί να χρησιμοποιηθεί για το σχεδιασμό numa-aware αλγορίθμων, εάν υλοποιηθεί με βάση τα χαρακτηριστικά των συστημάτων NUMA. Οι δομές δεδομένων που μπορούν να επωφεληθούν από αυτή την τεχνική είναι οι ουρές προτεραιότητας, στοίβες, ουρές FIFO κλπ., δομές δηλαδή που δεν παραλληλοποιούνται εύκολα.

1.5.2 Combining

Θα παρουσιάσουμε την τεχνική Flat Combining όπως προτείνεται στο [19]. Η βασική ιδέα της τεχνικής παρουσιάζεται στην εικόνα 1.7 και είναι ότι αντί η δομή δεδομένων να προστατεύεται με κάποιο fine-grained μηχανισμό κλειδώματος, υπάρχει το νήμα-combiner, το οποίο αναλαμβάνει να εκτελέσει ένα σύνολο λειτουργιών στη δομή εκ μέρους και των υπολοίπων νημάτων.



Σχήμα 1.7: Σχεδιασμός παράλληλης δομής δεδομένων με χρήση της τεχνικής flat combining

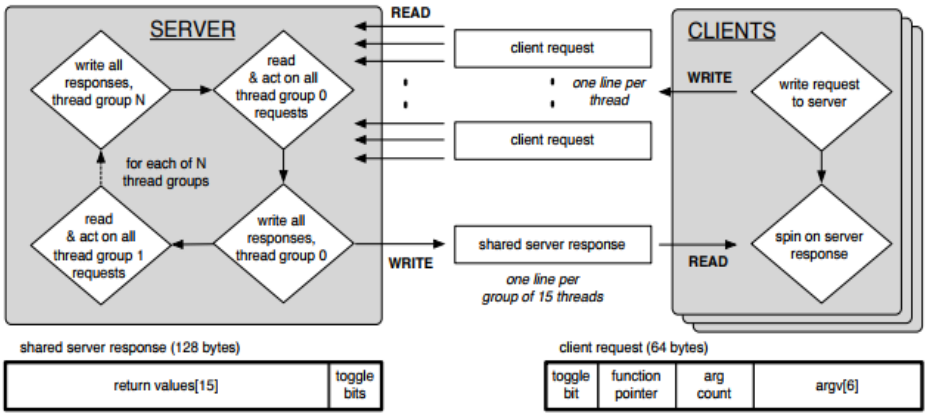
Μια υλοποίηση Flat Combining αποτελείται από μια σειριακή δομή δεδομένων η οποία προστατεύεται με κάποιο lock, και μια δυναμική λίστα δημοσιεύσεων (publication list) μεγέθους k*N όπου N ο αριθμός των νημάτων. Κάθε νήμα που πρόκειται να εκτελέσει μια λειτουργία στη δομή, δημοσιεύει το αίτημά του σε συγκεκριμένη θέση στη λίστα. Στη συνέχεια, ελέγχει εάν το lock είναι ελεύθερο και αν ναι επιχειρεί να το πάρει με χρήση κάποιας ατομικής εντολής. Το νήμα που θα καταφέρει να πάρει τελικά το lock γίνεται ο combiner. Ο combiner ελέγχει τη λίστα δημοσιεύσεων για αιτήματα που δεν έχουν πραγματοποιηθεί ακόμα, τα συλλέγει, εκτελεί τις αντίστοιχες λειτουργίες στη δομή δεδομένων, γράφει τα αποτελέσματα πίσω στη λίστα και αφήνει το lock. Τα υπόλοιπα νήματα, που δεν κατάφεραν να γίνουν combiner, ”περιμένουν” στις αντίστοιχες θέσεις στη λίστα για την απάντησή τους. Είναι σημαντικό, κατά το σχεδιασμό μιας δομής με χρήση Flat Combining, ο προγραμματιστής να μεριμνήσει ώστε να μην υπάρχουν φαινόμενα λιμοκτονίας λόγω καθυστέρησης του combiner.

Η τεχνική αυτή μπορεί να χρησιμοποηθεί για το σχεδιασμό numa-aware δομών δεδομένων κυρίως FIFO ουρών, στοιβών κλπ., δηλαδή σε περιπτώσεις που οι fine-grained μηχανισμοί προκαλούν μεγάλη κίνηση στο δίκτυο και δεν αποδίδουν επαρκώς. Απο την άλλη, σε δομές όπως σωροί, δυαδικά δένδρα κλπ. που χαρακτηρίζονται από υψηλά επίπεδα παραλληλισμού, οι fine-grained υλοποιήσεις παρουσιάζουν καλύτερα αποτελέσματα.

1.6 Fast Fly-Weight Delegation (FFWD)

Θα παρουσιάσουμε σε αυτό το σημείο το *ffwd* ([29]), μια τεχνική βασισμένη στην ιδέα του delegation που στοχεύει στη μείωση του κόστος της επικοινωνίας μεταξύ clients και server.

Η βασική ιδέα του delegation, όπως είπαμε, είναι ότι μόνο ένα νήμα μπορεί να έχει πρόσβαση στη δομή. Το γεγονός αυτό, παρόλο που μειώνει τον ανταγωνισμό μεταξύ των νημάτων και εξαλείφει την ανάγκη ύπαρξης μηχανισμού συγχρονισμού στη δομή δεδομένων, δεν επιτρέπει υψηλά επίπεδα παραλληλισμού. Για να επιτευχθεί, λοιπόν, όσο το δυνατόν μεγαλύτερη απόδοση, θα πρέπει να ανταλλάσονται παράλληλα πολλαπλά ζεύγη αιτημάτων/απαντήσεων ανάμεσα σε clients και server. Η επικοινωνία μεταξύ τους θα πρέπει να γίνεται όσο το δυνατόν πιο αποδοτικά και με γνώμονα τα πρωτόκολλα διατήρησης συνοχής μνήμης.



Σχήμα 1.8: Fast Fly-Weight Delegation: Το overhead της επικοινωνίας μεταξύ server και clients μειώνεται χρησιμοποιώντας συγκεκριμένο αριθμό από cache lines.

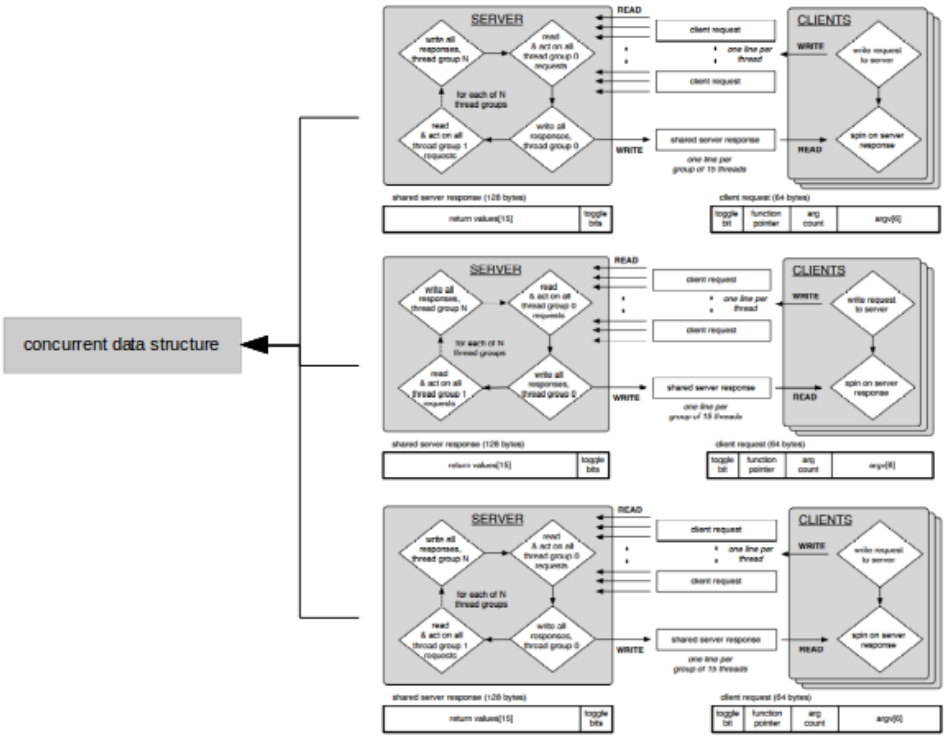
Η λειτουργία του ffwd παρουσιάζεται στην εικόνα 1.8. Η επικοινωνία μεταξύ server και clients γίνεται μέσω εγγραφής και ανάγνωσης σε μοιραζόμενες cache lines. Ο κάθε client δεσμεύει μια cache line στην οποία δημοσιεύει το αίτημά του. Ο server διαβάζει τα αιτήματα αυτά, εκτελεί τις αντίστοιχες λειτουργίες στη δομή και γράφει πίσω τις αντίστοιχες απαντήσεις. Για να μειωθεί ο αριθμός των μηνυμάτων που ανταλλάσσονται, ο server απαντά σε περισσότερα από ένα νήματα με την ίδια cache line. Σε ένα NUMA σύστημα, τα νήματα που μοιράζονται την ίδια απάντηση θα πρέπει ιδανικά να βρίσκονται στον ίδιο NUMA κόμβο. Ο server λαμβάνει και εκτελεί στη δομή δεδομένων όλα τα αιτήματα που προέρχονται από το ίδιο client group, αποθηκεύει τοπικά προσωρινά τις απαντήσεις μέχρι να μην υπάρχουν αιτήματα που εκρεμούν, γράφει τη συνολική απάντηση για όλη την ομάδα των clients και συνεχίζει με το επόμενο client group.

Το ffwd βελτιστοποιεί την τεχνική του delegation μειώνοντας το κόστος της επικοινωνίας μεταξύ client και server. Για το σκοπό αυτό, μπορεί να χρησιμοποιηθεί για το σχεδιασμό numa-aware δομών δεδομένων. Ο κάθε client χρησιμοποιεί μια cache line για να δημοσιεύεσει τα request του και άρα ο μόνος ανταγωνισμός προκαλείται από το server που διαβάζει κάθε request. Επίσης, καθώς η απάντηση σε κάθε NUMA κόμβο δίνεται συνολικά, αρκεί ένα νήμα να διαβάσει την αντίστοιχη cache line και να τη μεταφέρει τοπικά. Έτσι, μειώνεται ο αριθμός των cache line invalidations.

Το ffwd μπορεί να χρησιμοποιηθεί σε περιπτώσεις έντονου ανταγωνισμού μεταξύ των νημάτων και να έχει πολύ καλύτερη απόδοση από γνωστές μας lock-based ή lock-free τεχνικές. Παρόλα αυτά, δεδομένου του ότι μόνο ένας server μπορεί να έχει πρόσβαση στη δομή, η απόδοση του ffwd περιορίζεται. Συνεπώς ταιριάζει καλύτερα σε δομές που είναι πιο αποδοτικές όταν εκτελούνται με ένα νήμα παρά παράλληλα. Σε περιπτώσεις υψηλού παραλληλισμού, οι fine-grained τεχνικές έχουν καλύτερη απόδοση.

1.7 Numa Node Delegation (NUDDLE)

Το πρωτόκολλο επικοινωνίας που προτείνει το ffwd αξιοποιεί ένα συγκεκριμένο αριθμό από cache lines, ελαχιστοποιώντας το κόστος της επικοινωνίας μεταξύ clients και server και μειώνοντας έτσι την περιττή κίνηση στο δίκτυο διασύνδεσης. Παράλληλα, μόνο ένα νήμα έχει πρόσβαση στη δομή δεδομένων γι'αυτό και η δομή παραμένει στην ιεραρχία μνήμης του server, εξαλείφοντας έτσι την ανάγκη για συγχρονισμό μεταξύ των νημάτων. Παρόλα αυτά, η απόδοση του ffwd περιορίζεται από το ρυθμό με τον οποίο μπορεί ένα νήμα να διαχειριστεί τα εισερχόμενα αιτήματα και να εκτελέσει τις αντίστοιχες λειτουργίες.



Σχήμα 1.9: NUDDLE: οι servers δέχονται τα αιτήματα των clients και τα εκτελούν σε μια **παράλληλη** δομή δεδομένων

Για να ξεπεράσουμε τον περιορισμό αυτό, επεκτείναμε το ffwd και προτείνουμε το

Numa Node Delegation (NUDDLE) το οποίο μπορεί να χρησιμοποιηθεί για να μετατρέψει μια οποιαδήποτε μη numa-aware υλοποίηση - θα ονομάζουμε τις υλοποιήσεις αυτές ως numa-oblivious - στην αντίστοιχη numa-aware. Η βασική ιδέα του NUDDLE παρουσιάζεται στην εικόνα 1.9 και είναι ότι χρησιμοποιούνται πολλαπλοί servers από τον ίδιο numa κόμβο για να επεξαργαστούν τα αιτήματα των clients. Για να αποφευχθεί κατά το δυνατόν το numa effect, θα πρέπει τα νήματα που έχουν πρόσβαση στη δομή να παραμένουν σε ένα numa κόμβο. Έτσι, και η δομή παραμένει στον ίδιο numa κόμβο και έτσι όλες οι σχετικές με τη δομή προσβάσεις μνήμης είναι τοπικές. Για το σκοπό αυτό, θεωρούμε ότι όταν τρέχει το NUDDLE σε ένα numa σύστημα, ο μέγιστος αριθμός servers θα ισούται με τον αριθμό πηρύνων ενός numa node. Οι υπόλοιποι πυρήνες του συστήματος μπορούν να χρησιμοποιηθούν για client νήματα. Οι servers χρησιμοποιούν το πρωτόκολλο ανταλλαγής μηνυμάτων του ffwd για να επικοινωνήσουν με τους clients και εκτελούν τα requests τους στη δομή.

Εφόσον, λοιπόν, υπάρχουν περισσότεροι του ενός servers που έχουν πρόσβαση στη δομή, θα πρέπει να χρησιμοποιηθεί κάποιος μηχανισμός συγχρονισμού. Συνεπώς, σε αντίθεση με το ffwd, το Nuddle χρειάζεται παράλληλη υλοποίηση της δομής ώστε να μην παραβιαστεί η ορθότητά της. Ο μηχανισμός Nuddle μπορεί να χρησιμοποιηθεί με οποιαδήποτε παράλληλη υλοποίηση είτε lock-based είτε lock-free. Στην παρούσα διπλωματική ασχοληθήκαμε με ουρές προτεραιότητας και χρησιμοποιήσαμε τη numa-oblivious υλοποίηση που προτείνεται στο [2]. Πρόκειται για μια relaxed, lock-free ουρά προτεραιότητας, σύμφωνα με την οποία λειτουργία *deleteMin()* επιστρέφει ένα στοιχείο ανάμεσα στα $O(p*(logp)^3)$ μικρότερα στοιχεία της δομής, όπου p ο αριθμός των νημάτων, μειώνοντας έτσι τον ανταγωνισμό μεταξύ των νημάτων και προσφέροντας υψηλότερα επίπεδα παραλληλισμού. Η συγκεκριμένη υλοποίηση επιλέχτηκε καθώς παρουσίασε την πιο αποδοτική συμπεριφορά στα πειράματά μας. Παρόλα αυτά, το Nuddle είναι μια black-box τεχνική, μπορεί δηλαδή να χρησιμοποιηθεί με οποιαδήποτε παράλληλη υλοποίηση. Στην ουσία το Nuddle λειτουργεί ως ένα Numa-aware πλαίσιο και δεν μεταβάλει το πώς προσπελαύνεται η δομή δεδομένων, αλλά το ποιοί έχουν πρόσβαση σε αυτή. Δεν διαταράσσει, συνεπώς, ούτε την ορθότητα ούτε την πρόοδο της numa-oblivious υλοποίησης που χρησιμοποιείται

1.7.1 Πειραματική Αξιολόγηση

Στη συνέχεια, αξιολογήσαμε πειραματικά την απόδοση του Nuddle και το συγκρίναμε τόσο με numa-oblivious (βιβλιοθήκη Ascylib [10]) όσο και με numa-aware ουρές προτεραιότητας. Για περισσότερες λεπτομέρειες και για μια πληρέστερη εικόνα των πειραμάτων ο αναγνώστης μπορεί να ανατρέξει στο κεφάλαιο 4. Εδώ θα παρουσιάσουμε ενδεικτικά και θα σχολιάσουμε τα αποτελέσματα των πειραμάτων σε ένα Numa σύστημα τα στοιχεία του οποίου παρουσιάζονται στον πίνακα 1.1. Στο εξής θα αναφερόμαστε στο συγκεκριμένο σύστημα με το όνομα sandman.

Για τα αποτελέσματα που θα παρουσιάσουμε στη συνέχεια ισχύουν τα εξής:

- Η αξιολόγηση των διάφορων υλοποιήσεων έγινε με βάση το throughput το οποίο μετράμε σε εκατομύρια λειτουργίες/δευτερόλεπτο. Οι λειτουργίες που μπορούν να εκτελέσουν τα νήματα στα πειράματά μας είναι: insert(key),lookup(key),deleteMin().

Reference name	**sandman**
CPU	4 * Intel Xeon E5-4620
Processor Base Frequency	2.20 GHz
Number of Cores	4 * 8
Number of Threads	4 * 16
L1(data) cache/core	32 KB, 8-way, 64b block size
L2 cache/core	256 KB, 8-way, 64b block size
L3 cache/socket	16 MB, 16-way, 64b block size
RAM	256 GB
OS	Debian 8.8
Linux kernel	4.0.4

Πίνακας 1.1: Τεχνικά χαρακτηριστικά της NUMA πλατφόρμας που χρησιμποποιήθηκε για τα πειράματα.

Το key είναι ένας ακέραιος αριθμός μεγαλύτερος του 1 και μικρότερος του ”εύρους”, όπου σαν εύρος ορίζουμε το διπλάσιο του αρχικού μεγέθους της δομής. Σε κάθε λειτουργία το key επιλέγεται τυχαία. Ανάμεσα σε δύο συνεχόμενες λειτουργίες μεσολαβεί ένα διάστημα απραξίας μεγέθους 25 κύκλων για κάθε νήμα.

- Για να έχουμε μια καλύτερη εικόνα των υπό εξέταση υλοποιήσεων, πειραματιστήκαμε με το αρχικό μέγεθος της δομής, τον αριθμό των νημάτων που μπορούν να την προσπελάσουν, και το είδος και το ποσοστό των λειτουργιών που τα νήματα αυτά εκτελούν στη δομή.

- Όπως αναφέραμε, στο πρωτόκολλο επικοινωνίας του FFWD (άρα και στο Nuddle) ο server χρησιμοποιεί μία cache line για να απαντήσει σε μια ομάδα από clients. Ο αριθμός των clients που μοιράζονται την ίδια απάντηση εξαρτάται, συνεπώς, από το μέγεθος της cache line. Σε ένα block απάντησης υπάρχουν N τμήματα των 8 bytes, όπου N ο αριθμός των clients που ανήκουν στο ίδιο response group, καθώς και ένα επιπλέον πεδίο των 8 bytes που ”ενημερώνει” τους clients αν είναι έτοιμη η απάντηση. Το μέγεθος της cache line στην πλατφόρμα sandman είναι 64 bytes, και άρα στα πειράματά μας 7 clients μοιράζονται την ίδια απάντηση.

- Σε κάθε εκτέλεση τα νήματα γίνονται pin σε συγκεκριμένους πηρύνες. Στην περίπτωση του sandman ο κάθε numa node έχει 8 πυρήνες και 16 hardware threads. Παράλληλα, σύμφωνα με το FFWD υπάρχει 1 server ενώ στο NUDDLE έχουμε ο-

ρίσει ότι οι servers θα είναι όσοι και οι πυρήνες ενός numa node άρα 8 σε αυτή την περίπτωση. Εάν ονομάσουμε με s0,s1,s2,s3 τα τέσσερα numa nodes του sandman, τότε για τις υπο εξέταση υλοποιήσεις ισχύει ότι:

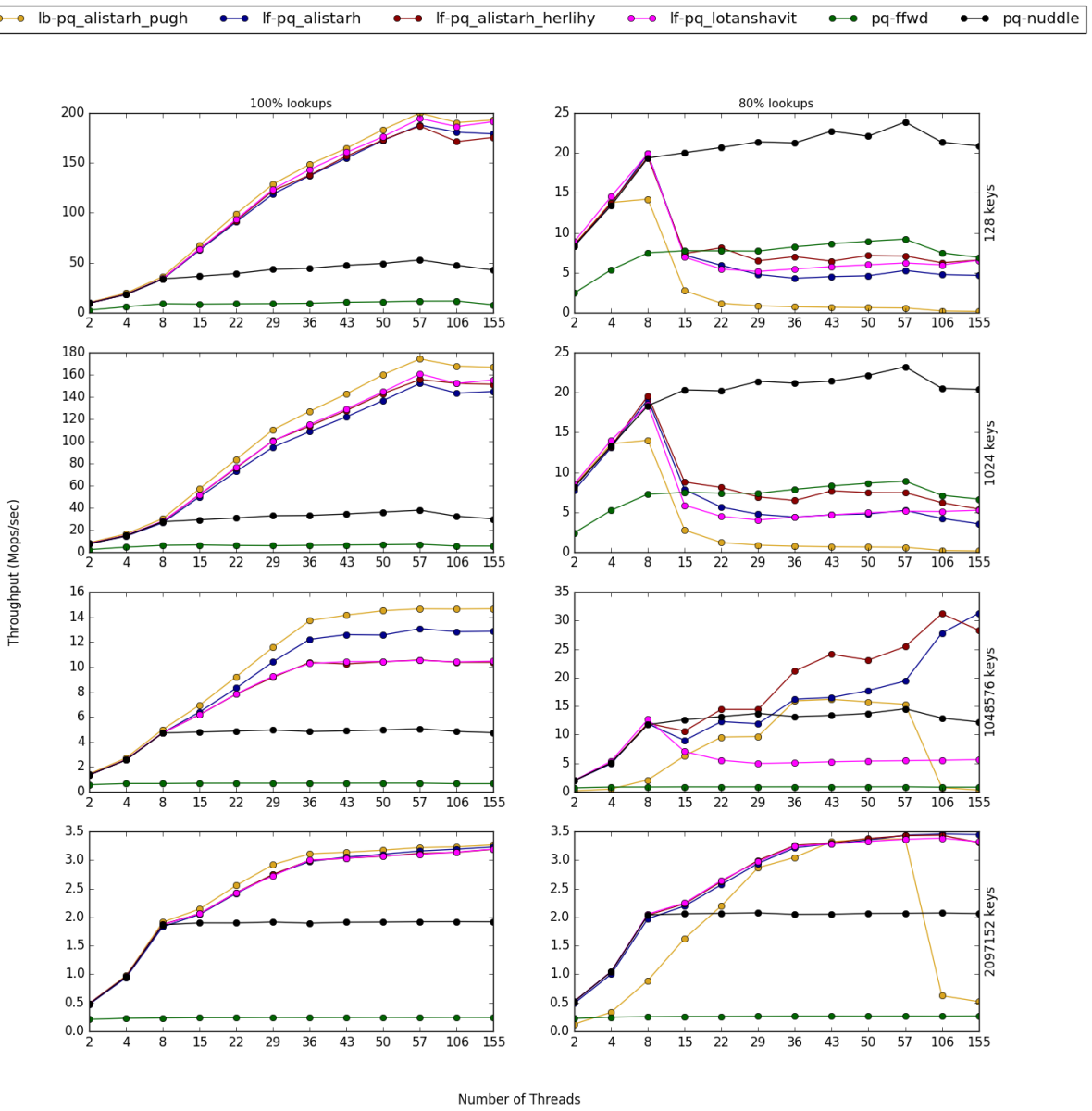
- εάν $N \leq 8 \Rightarrow$ N νήματα τοποθετούνται στους 8 πυρήνες του s0. Στην περίπτωση του FFWD 1 από αυτά λειτουργεί ως server, ενώ στην περίπτωση του NUDDLE και τα N νήματα έχουν ρόλο server.
- εάν $29 \leq N > 8 \Rightarrow$ 8 νήματα γίνονται pin στο s0, the επόμενα 7 στο s1, τα επόμενα 7 στο s2, και τέλος τα επόμενα 7 στο s3.
- εάν $57 \leq N > 29 \Rightarrow$ τα πρώτα 29 νήματα κατανέμονται σύμφωνα με τον τρόπο που περιγράφεται παραπάνω ενώ για τα επόμενα N-29 νήματα χρησιμοποείται hyperthreading , τοποθετώντας 7 νήματα σε καθένα από τα 4 numa nodes.
- εάν $N > 57 \Rightarrow$ πάνω από δύο νήματα γίνονται pin σε κάποιους από τους πηρύνες του μηχανήματος (oversubscription effect) με τον τρόπο που αναφέρουμε στα τρία παραπάνω σημεία.

Παρατηρώντας στις εικόνες 1.10 και 1.11 από τα αποτελέσματα των πειραμάτων μας μπορούμε να συμπεράνουμε τα εξής:

- Οι numa-oblivious παράλληλες υλοποιήσεις που εμπεριέχονται στη βιβλιοθήκη Ascylib κλιμακώνουν ικανοποιητικά σε περιπτώσεις όπου ο ανταγωνισμός μεταξύ των νημάτων δεν είναι έντονος όπως για παράδειγμα σε benchmark με 100% lookup λειτουργίες ή όταν το μέγεθος της δομής είναι σχετικά μεγάλο (1048576 και 2097152 στοιχεία). Όταν ο ανταγωνισμός αυτός αυξάνεται, κυρίως με την αύξηση του ποσοστού των deleteMin λειτουργιών, το numa effect γίνεται πιο έντονο και καθώς παρατηρούμε στις αντίστοιχες περιπτώσεις, η απόδοση των numa-oblivious υλοποιήσεων μειώνεται δραματικά όταν ο αριθμός των νημάτων υπερβαίνει τον αριθμό των πηρύνων ενός numa node (8 σε αυτή την περίπτωση).

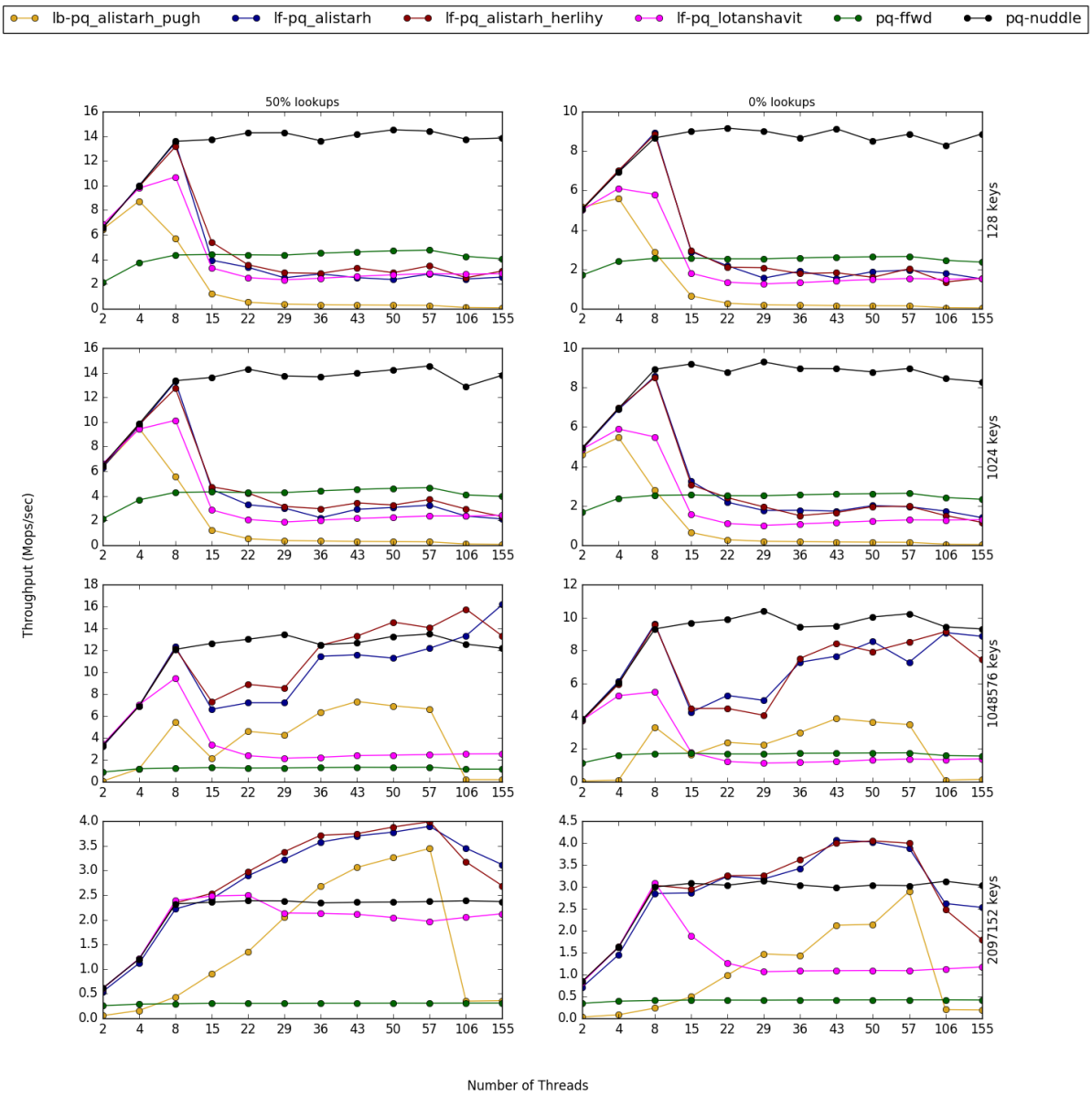
- Η απόδοση του FFWD είναι σταθερή καθώς μεταβάλλεται ο αριθμός των νημάτων και περιορίζεται από το γεγονός ότι μόνο ένα νήμα, ο server, μπορεί να προσπελάσει τη δομή. Παρατηρούμε ότι ο μηχανισμός επικοινωνίας μεταξύ clients-server προσθέτει αμελητέο εώς καθόλου overhead στην εκτέλεση. Το FFWD παρουσιάζει πιο αποδοτική συμπεριφορά από τις υλοποιήσεις του Ascylib μόνο σε περιπτώσεις πολύ υψηλού ανταγωνισμού μεταξύ των νημάτων (μικρό μέγεθος δομής, υψηλό ποσοστό deleteMin λειτουργιών)

- Ο αλγόριθμος NUDDlE κλιμακώνει όσο ο αριθμός των νημάτων είναι μικρότερος από 8 (μέγιστος αριθμός servers). Στη συνέχεια, καθώς ολοένα και περισσότεροι clients προστίθενται, η απόδοση του NUDDLE παραμένει σταθερή επαληθεύοντας και εδώ ότι το πρωτόκολο επικοινωνίας που χρησιμοποείται δεν προσθέτει κάποιο overhead στην εκτέλεση. Το NUDDLE δεν επηρρεάζεται από το numa effect και



Σχήμα 1.10: Αξιολόγηση παράλληλων ουρών προτεραιότητας στο sandman για διάφορα μεγέθη δομής και ποσοστά λειτπυργιών. Ο Χ άξονας αναπαριστά τον αριθμό των νημάτων ενώ ο Υ το throughput των υλοποιήσεων (Million Operations/second).

παρουσιάζει πιο αποδοτική συμπεριφορά όταν ο ανταγωνισμός μεταξύ των νημάτων είναι υψηλός (μικρή δομή, μεγάλο ποσοστό update λειτουργιών), ενώ σε περιπτώσεις υψηλού παραλληλισμού, οι υλοποιήσεις του Ascylib αποδεικνύονται καλύτερες.



Σχήμα 1.11: Αξιολόγηση παράλληλων ουρών προτεραιότητας στο sandman. Το εύρος των στοιχείω σε κάθε πείραμα είνα διπλάσιο του αρχικού μεγέθους της δομής.

- Το φαινόμενο του oversubscription φαίνεται να επηρρεάζει τις υλοποιήσεις του ascylib σε ορισμένες περιπτώσεις, άλλοτε αυξάνοντας και άλλοτε μειώνοντας την απόδοσή τους, αλλά όχι τους αλγορίθμους FFWD και NUDDLE των οποίων το throughput παραμένει σταθερό.

1.8 SmartPQ

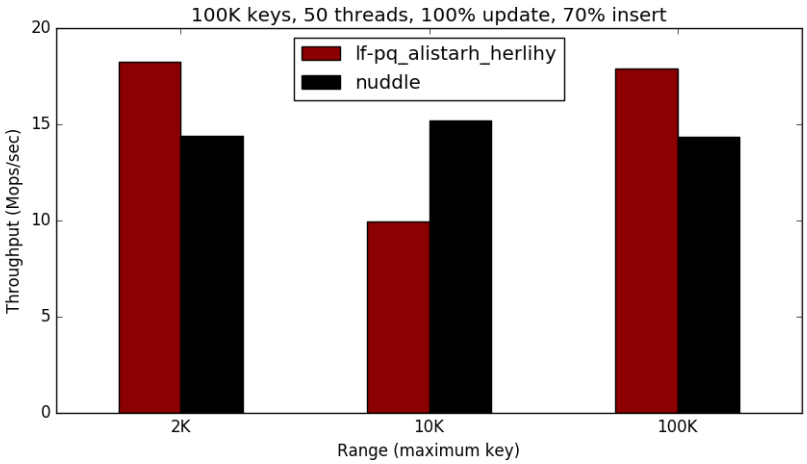
1.8.1 Η ανάγκη για μια "έξυπνη" ουρά προτεραιότητας

Τα παραπάνω αποτελέσματα κάνουν εμφανές το γεγονός ότι κατά το σχεδιασμό μιας numa-aware ουράς προτεραιότητας, υπάρχει ένα tradeoff ανάμεσα στο υψηλό επίπεδο παραλληλισμού που χαρακτηρίζει μια numa-oblivious υλοποίηση και στην τοπικότητα των δεδομένων που προσφέρει μια αντίστοιχη numa-aware. Απομακρυσμένες πρσοβάσεις στη μνήμη και διαρκείς invalidations των cache lines οδηγούν σε μείωση της απόδοσης παράλληλων numa-oblivious αλγορίθμων ειδικά σε περιπτώσεις υψηλού ανταγωνισμού μεταξύ των νημάτων. Από την άλλη, οι numa-aware αλγόριθμοι στοχεύουν στο να μειώσουν τον αριθμό των απομακρυσμένων προσβάσεων στη μνήμη και να εκμεταλλευτούν στο έπακρο την τοπικότητα ενός numa node, μειώνοντας έτσι τον αριθμό των νημάτων που μπορούν να προσπελάσουν τη δομή ταυτόχρονα και περιορίζοντας έτσι τον παραλληλισμό.

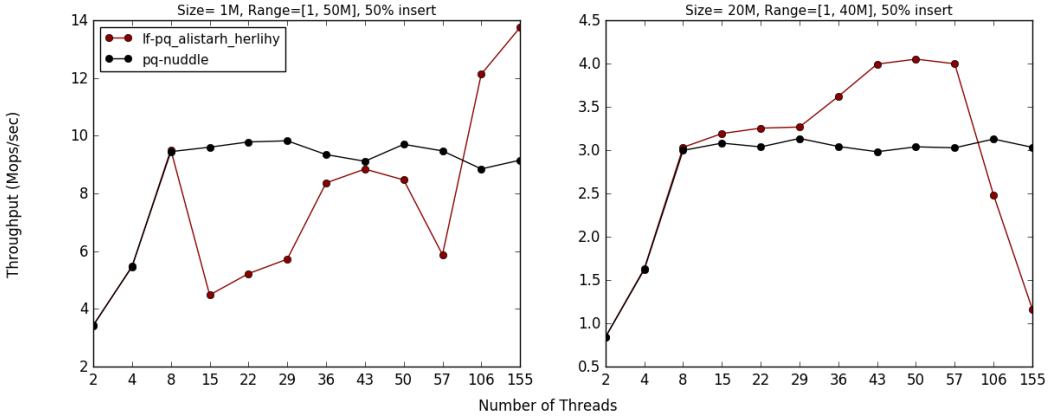
Το γεγονός αυτό μας οδηγεί στο συμπέρασμα ότι όταν κερδίζουμε σε παραλληλισμό, χάνουμε σε τοπικότητα δεδομένων και αντίστροφα. Το συμπέρασμα αυτό φαίνεται από τα πειράματά μας και δείχνει ξεκάθαρα ότι η απόδοση μιας ουράς προτεραιότητας εξαρτάται από πολλούς παράγοντες, όπως η πλατφόρμα και οι συνθήκες στις οποίες εκτελείται. Ιδανικά, θα θέλαμε να υπήρχε μια υλοποίηση η οποία να παρουσιάζει την υψηλότερη απόδοση από όλες τις άλλες ανεξαρτήτως μηχανήματος και συνθηκών εκτέλεσης. Προς αυτή την κατεύθυνση μελετήσαμε στην παρούσα διπλωματική τον τρόπο με τον οποίο επηρρεάζεται μια ουρά προτεραιότητας από το περιβάλλον στο οποίο εκτελείται και σχεδιάσαμε μια υλοποίηση η οποία διατηρεί τη μέγιστη δυνατή απόδοση υπό όλες τις συνθήκες εκτέλεσης. Επικεντρωθήκαμε στο μηχάνημα sandman για τα πειράματα και το σχεδιασμό μας, παρόλα αυτά πιστεύουμε ότι η μεθοδολογία που περιγράφουμε στη συνέχεια μπορεί να αξιοποιηθεί και σε άλλες numa πλατφόρμες.

Οι κύριοι παράγοντες που επηρρεάζουν, λοιπόν, την απόδοση μιας ουράς προτεραιότητας σε ένα Numa σύστημα είναι: το μέγεθος της δομής, το εύρος των στοιχείων που εισάγονται ή αναζητούνται στη δομή, το πλήθος των νημάτων που προσπελαύνουν την ουρά καθώς και το είδος και το ποσοστό των λειτουργιών που εκτελούνται στη δομή (ποσοστό insert(key), deleteMin(), lookup(key) λειτουργιών). Μελετήσαμε τον τρόπο με τον οποίο οι παράγοντες αυτοί σχετίζονται με την απόδοση είτε numa-oblivious είτε numa-aware υλοποιήσεων όπως ο αλγόριθμος alistarh-herlihy[2] και ο αλγόριθμος NUDDLE αντίστοιχα, και προσπαθήσαμε να μοντελοποιήσουμε τη συσχέτιση αυτή με κάποιο μαθηματικό ή στατιστικό τρόπο. Παρόλα αυτά, καταλήξαμε στο ότι αυτό είναι αδύνατον καθώς η σχέση που μελατάμε είναι ιδιαιτέρως πολύπλοκη και δε μπορεί εύκολα να μοντελοποιηθεί.

Στο διάγραμμα 1.12 συγκρίνουμε τις δύο υλοποιήσεις μεταβάλλοντας το εύρος των στοιχείων που εισάγουμε στη δομή. Παρατηρούμε ότι καθώς το εύρος αυξάνεται από 2K σε 10K, η απόδοση της numa-oblivious υλοποίησης μειώνεται και η numa-aware αποδεικνύεται καλύτερη. Παρόλα αυτά, μια περαιτέρω αύξηση στο εύρος οδηγεί και πάλι σε αύξηση του throughput της alistarh-herlihy υλοποίησης. Επιπλέον, στα διαγράμματα της



Σχήμα 1.12: Πειραματική αξιολόγηση του NUDDLE και της Alistarh-Herlihy ουράς προτεραιότητας. Το αρχικό μέγεθος της δομής είναι 100000, ενώ το ποσοστό insert και deleteMin λειτουργιών είναι 70% και 30% αντίστοιχα. Το εύρος των στοιχείων που εισάγονται στη δομή ποικίλει από 2K σε 100K.



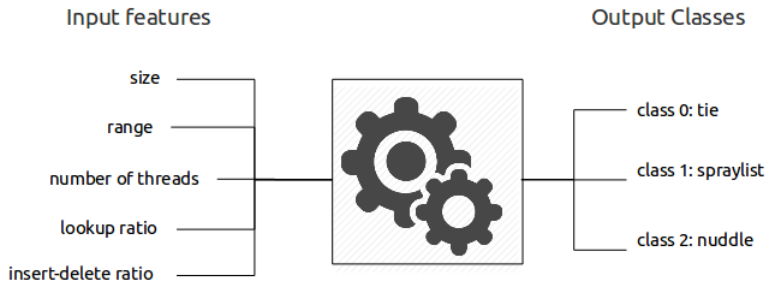
Σχήμα 1.13: Πειραματική αξιολόγηση της ουράς alistarh-herlihy και του NUDDLE σε ένα benchmark με 50% insert και 50% deleteMin(). Το φαινόμενο του oversubscription επηρεάζει την απόδοση του alistarh-herlihy.

εικόνας 1.13 μεταβάλλουμε τον αριθμό των νημάτων και παρατηρούμε τον τρόπο με τον οποίο επηρρεάζουν τις δύο υλοποιήσεις. Βλέπουμε ότι το φαινόμενο του oversubscription - δηλαδή όταν ο αριθμός των νημάτων είναι μεγαλύτερος από 57 - δε φαίνεται να επηρρεάζει την απόδοση του Nuddle αλλά μεταβάλει τη συμπεριφορά της alistarh-herlihy υλοποίησης άλλοτε αυξάνοντας και άλλοτε μειώνοντας την απόδοσή της. Συμπεραίνουμε λοιπόν ότι δεν μπορούμε να προβλέψουμε πάντα την επίδραση των διαφόρων παραμέτρων εκτέλεσης στην απόδοση των υλοποιήσεων και άρα να καθορίσουμε εύκολα τον τρόπο με τον οποίο θα λειτουργεί μια ”έξυπνη” ουρά προτεραιότητας. Για το σκοπό αυ-

τό, χρησιμοποιήσαμε τεχνικές μηχανικής μάθησησης για να μοντελοποιήσουμε αυτή την επίδραση.

1.8.2 Σχεδιασμός της SmartPQ με χρήση Μηχανικής Μάθησης

Όπως έχει γίνει φανερό από τα παραπάνω, μια ουρά προτεραιότητας μπορεί να εκτελεσθεί σε ένα NUMA σύστημα είτε με numa-aware είτε με numa-oblivious τρόπο. Το πρώτο βήμα για το σχεδιασμό μιας ”έξυπνης” ουράς προτεραιότητας ήταν να μοντελοποιήσουμε το πρόβλημα της επιλογής του καταλληλότερου τρόπου εκτέλεσης ως ένα πρόβλημα ταξινόμησης ώστε να μπορέσει να επιλυθεί με τεχνικές μηχανικής μάθησης. Στόχος μας ήταν να κατασκευάσουμε ένα μοντέλο το οποίο, δοθέντων συγεκριμένων γνωρισμάτων εισόδου, δηλαδή δοθέντων συνθηκών εκτέλεσης, να ”επιλέγει” την πιο αποδοτική υλοποίηση ανάμεσα στη δική μας numa-aware Nuddle και στη numa-oblivious ουρά προτεραιότητας που έχει προταθεί από τον Alistarh [2] την οποία θα αναφέρουμε στο εξής είτε ως alistarh-herlihy είτε ως spraylist. Συνεπώς, στόχος μας ήταν να δημιουργήσουμε ένα μοντέλο ταξινόμησης όπως αυτό της εικόνας 1.14, το οποίο να δέχεται ως είσοδο το αρχικό μέγεθος της δομής, το εύρος των στοιχείων, τον αριθμό των νημάτων, καθώς και τα ποσοστά lookup-insert-deleteMin, και να προβλέπει μία από τις τρεις κλάσεις εξόδου. Οι κλάσσεις 1 και 2 αντιπροσωπεύουν τους numa-oblivious και numa-aware τρόπους εκτέλεσης αντίστοιχα, ενώ η κλάση 0 αντιστοιχεί σε περιπτώσεις όπου οι δύο αλγόριθμοι έχουν παρόμοια απόδοση. Θεωρήσαμε ότι οι δύο υλοποιήσεις έχουν παρόμοια συμπεριφορά όταν η απόλυτη τιμή της διαφοράς throughput μεταξύ τους είναι μικρότερη ή ίση από 1.5 Mops. Σε αυτές τις περιπτώσεις αναμένουμε ότι η έξοδος του μοντέλου μας είναι η ”ουδέτερη” κλάση 0.



Σχήμα 1.14: Μοντελοποιήσαμε το πρόβλημα της επιλογής του καλύτερου αλγορίθμου ως ένα πρόβλημα ταξινόμησης χρησιμοποιώντας 5 γνωωρίσματα εισόδου (features) και 3 κλάσεις.

Για την δημιουργία του παραπάνω μοντέλου, καθώς και για οποιοδήποτε μοντέλο μηχανικής μάθησης, ήταν απαραίτητη η δημιουργία ενός επαρκούς και αντιπροσωπευτικού συνόλου εκπαίδευσης. Για αυτό το λόγο, κατασκευάσαμε ένα σύνολο 50000 διαφορετικών συνδυασμών των features του προβλήματος, εκτελέσαμε στην πλατφόρμα sandman τους αλφορίθμους nuddle και alistarh-herlihy για κάθε συνδυασμό των παραπάνω features και καταγράψαμε την απόδοσή τους. Στη συνέχεια, ταξινομήσαμε τα αποτελέσματά στις 3 κλάσεις του προβλήματος, συγκρίνοτας, για κάθε συνδυασμό features το throughput των

δύο υλοποιήσεων. Πιο συγκεκριμένα, εάν T_{sp} και T_n είναι το throughput του spraylist και του nuddle αντίστοιχα, τότε:

$$diff = T_{sp} - T_n$$

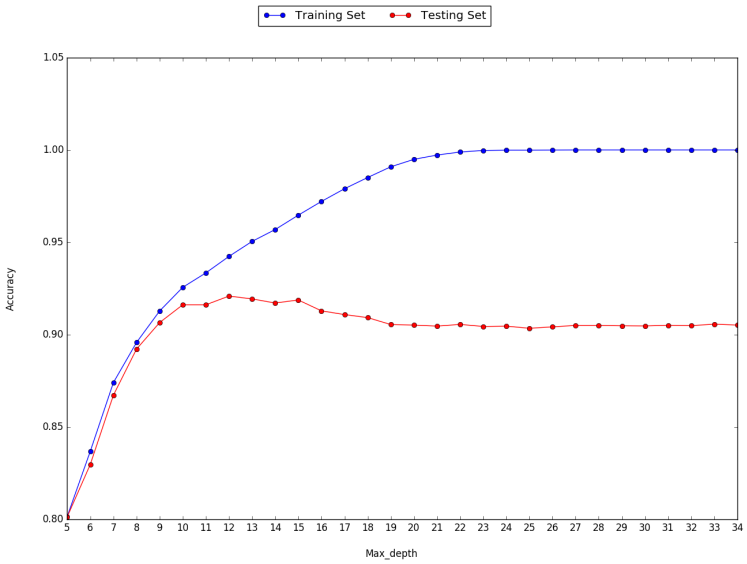
$$-1.5 \leq diff \leq 1.5 => 0$$
$$diff > 1.5 => 1$$
$$diff < -1.5 => 2$$

Με αυτό τον τρόπο κατασκευάσαμε το dataset για την εκπαίδευση και την αξιολόγηση του μοντέλου. Συγκεκριμένα, 80% του συνόλου δεδομένων χρησιμοποιήθηκε για την εκπαίδευση και 20% για την αξιολόγηση του αλγορίθμου. Για την επίλυση του προβλήματός μας χρησιμοποιήσαμε δέντρα αποφάσων (decision trees) καθώς είναι μοντέλα που μπορούν να ερμηνευθούν εύκολα και δεν απαιτούν σημαντική υπολογιστική ισχύ. Χρησιμοποιήσαμε το scikit-learn [25] μια βιβλιοθήκη της python για μηχανική μάθηση η οποία περιέχει υλοποιήσεις δέντρων απόφασης για προβλήματα ταξινόμησης και τα εκπαιδεύει μέσω του αλγορίθμου CART.

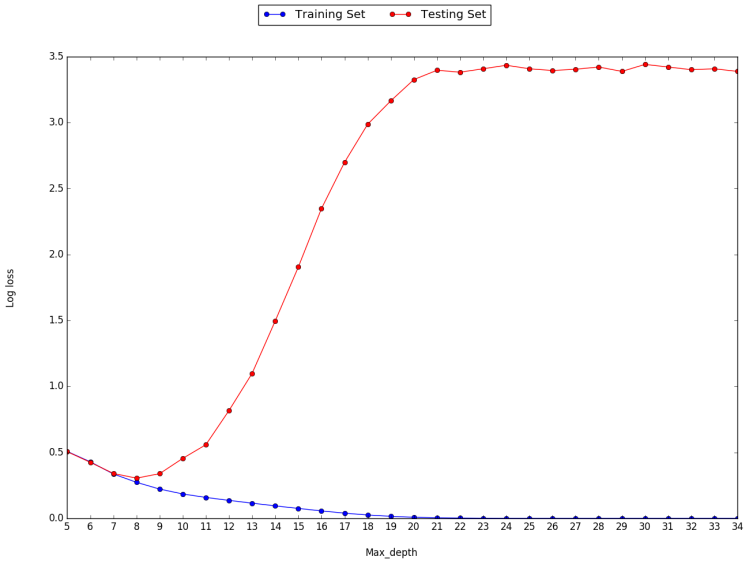
Αρχικά, κάναμε χρήση ενός decision tree ταξινομητή με τις προκαθορισμένες παραμέτρους. Παρά την αρκετά υψηλή ακρίβειά του (γύρω στο 90%), το βάθος του δέντρου ήταν μεγάλο και άρα ο κίνδυνος υπερπροσαρμογής στο σύνολο εκπαίδευσης ήταν ιδιαίτερα υψηλός. Για να βελτιώσουμε το μοντέλο και να αυξήσουμε την ακρίβειά του σε άγνωστα δεδομένα πειραματιστήκαμε με διάφορες παραμέτρους του ταξινομητή, όπως τα: criterion, max_depth, min_samples_split, min_samples_leaf. Εκτελέσαμε 10-fold cross validation αξιολογώντας το μοντέλο μας αρχικά σε σχέση με την ακρίβειά του. Καθώς όμως το σύνολο εκπαίδευσης δεν ήταν ισορροπημένο ανάμεσα στις 3 κλάσεις, η ακρίβεια (accuracy) δεν ήταν αντιπροσωπευτική μετρική. Για το σκοπό αυτό, χρησιμοποιήσαμε επίσης τις εξής μετρικές: precision, recall, f1-score, logarithmic loss. Επιπλέον, για να έχουμε μια καλύτερη εικόνα της διαδικασίας βελτιστοποίησης του μοντελου, οπτικοποιήσαμε την επίδραση των υπό αναζήτηση παραμέτρων στις διάφορες μετρικές. Ενδεικτικά δείχνουμε εδώ στα διαγράμματα 1.15 και 1.16 το accuracy και το logarithimc loss αντίστοιχα συναρτήσει της παραμέτρου max_depth τόσο για το σύνολο εκπαίδευσης όσο και για το σύνολο αξιολόγησης. Παρατηρούμε ότι για μικρές τιμές του max_depth το accuracy είναι πολύ χαμηλό, ενώ για μεγάλες τιμές της παραμέτρου αυξάνεται η λογαριθμική απώλεια των προβλέψεων στο σύνολο αξιολόγησης άρα αυτομάτως η τιμή του max_depth θα πρέπει να περιοριστεί στο διάστημα [10,13].

Ακολουθώντας αντίστοιχη διαδικασία και για τις υπόλοιπες παραμέτρους, και με χρήση της τεχνικής cross-validation που αναφέραμε παραπάνω επιλέξαμε τις παραμέτρους για το τελικό μοντέλο, οι οποίες φαίνονται στον πίνακα 1.2.

Στον πίνακα 1.3 παρουσιάζονται τα αποτελέσματα της αξιολόγησης του τελικού ταξινομητή ως προς διάφορες μετρικές στο αρχικό σύνολο αξιολόγησης (Test set 1). Παράλληλα, παρουσιάζεται και η ακρίβεια του ταξινομητή σε ένα σύνολο 11000 άγνωστων στοιχείων (Test Set 2), όπου το καθένα αντιπροσωπεύει ένα διαφορετικό συνδυασμό από features.



Σχήμα 1.15: Μέση ακρίβεια στα σύνολα εκπαίδευσης και εξέτασης συναρτήσει της παραμέτρου max_depth.



Σχήμα 1.16: Λογαριθμική απώλεια στα σύνολα εκπαίδευσης και εξέτασης για διαφορετικ τιμές της παραμέτρου max_depth.

Στη συνέχεια, μετατρέψαμε το σύνολο των κανόνων του decision tree ταξινομητή σε μια C συνάρτηση, ώστε να μπορεί να χρησιμοποιηθεί πιο εύκολα από την υλοποίησή μας. Η smart priority queue που προτείνουμε λαμβάνει ως είσοδο το αρχικό μέγεθος, το μέγιστο εύρος των στοιχείων, τον αριθμό των νημάτων καθώς και τα ποσοστά κατά τα οποία θα εκτελεσθούν οι λειτουργίες lookup-insert-deleteMin, καλεί την παραπάνω

Παράμετρος	Τιμή
min_samples_split	10
max_depth	13
criterion	entropy
min_samples_leaf	3

Πίνακας 1.2: Οι τιμές που προέκυψαν για τις παραμέτρους του ταξινομητή έπειτα από τη διαδικασία βελτιστοποίησης.

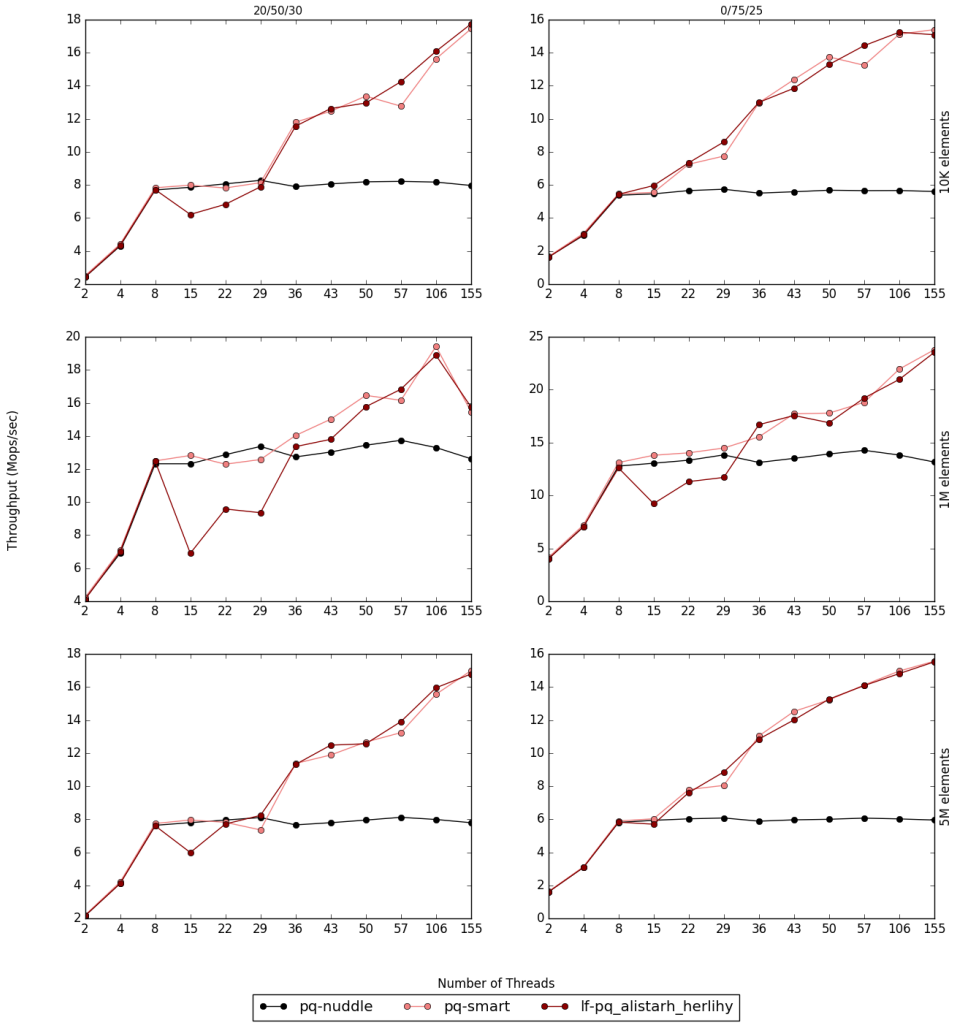
Test set 1					Test set 2
Accuracy	Precision	Recall	F1-score	Log Loss	Accuracy
0.934	0.914	0.919	0.916	0.666	0.891

Πίνακας 1.3: Αξιολόγηση του τελικού μοντέλου ως προς διάφορες μετρικές

συνάρτηση και εκτελείται σύμφωνα είτε με τον αλγόριθμο Nuddle είτε με τον αλγόριθμο alistarh-herlihy. Αξιολογήσαμε την ακρίβεια του ταξινομητή μας, και άρα την απόδοση της smart ουράς προτεραιότητας με ένα πλήθος από benchmark όπως φαίνεται ενδεικτικά και στην εικόνα 1.17. Εκεί η smart δομή συγκρίνεται με τις υλοποιήσεις alistarh-herlihy και nuddle και βλέπουμε ότι σε κάθε περίπτωση επιλέγει τον πιο αποδοτικό αλγόριθμο και άρα επιτυγχάνει την υψηλότερη δυνατή απόδοση.

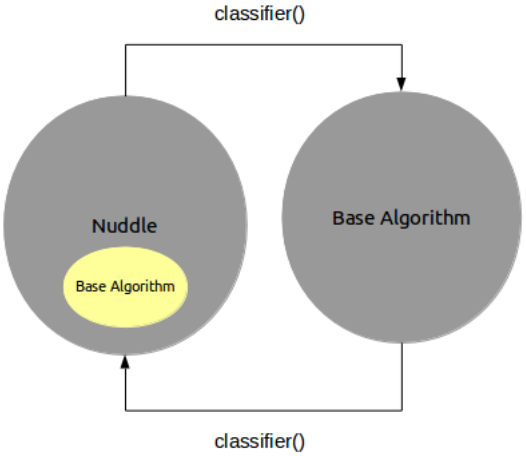
Σε μια εφαρμογή, όμως, τα features που περιγράφουμε παραπάνω μεταβάλλονται με το χρόνο, για παράδειγμα το πλήθος των νημάτων που εκτελούν λειτουργίες στη δομή καθώς και το είδος αυτών των λειτουργιών δεν παραμένουν σταθερά. Στη συνέχεια της διπλωματικής προσομοιώσαμε ένα περιβάλλον αυτής της μορφής, ένα δυναμικό δηλαδή περιβάλλον, και επεκτείναμε τη smart ουρά προτεραιότητας ώστε να μπορεί να εκτελεσθεί σε ένα τέτοιο περιβάλλον και να μεταβαίνει δυναμικά από τη numa-aware στη numa-oblivious υλοποίηση και αντίστροφα όταν αυτό χρειάζεται, όπως φαίνεται και στην εικόνα 1.18.

Η υλοποίηση μας βασίζεται στο Nuddle, το οποίο μπορεί να χρησιμοποιηθεί με οποιαδήποτε numa-oblivious ουρά προτεραιότητας και να τη μετατρέψει στην αντίστοιχη numa-aware χωρίς να αλλάξει τον τρόπο με τον οποίο πραγματοποιούνται οι βασικές λειτουργίες στη δομή (lookup, insert, deleteMin), όπως φαίνεται και στην εικόνα 1.18. Το nuddle δηλαδή χρησιμοποιεί τους ίδιους αλγορίθμους για τις 3 λειτουργίες στη δομή με τη numa-oblivious υλοποίηση, αλλά ορίζει ότι μόνο όσα νήματα είναι servers μπορούν να εκτελέσουν τους συγκεκριμένους αλγόριθμους: τα υπόλοιπα νήματα (clients) θα πρέπει, για να εκτελέσουν μία από τις παραπάνω λειτουργίες, να κάνουν delegate το αίτημά τους στους servers. Συνεπώς, χρησιμοποιώντας μια numa-oblivious ουρά προτεραιότητας (την υλοποίηση alistarh-herlihy στην περίπτωσή μας), το Nuddle και τον ταξινομητή που περιγράψαμε παραπάνω, επεκτείναμε τη smart ουρά προτεραιότητας ως εξής: κατά τη διάρκεια της εκτέλεσης, κάθε φορά που υπάρχει μια αλλαγή στα υπό εξέταση features, η smart υλοποίηση καλεί τη συνάρτηση που περιέχει τους κανόνες του decision tree ταξινομητή με τα συγκεκριμένα features ως ορίσματα και ανάλογα με το αποτέλεσμα της



Σχήμα 1.17: Πειραματική αξιολόγηση των υλοποιήσεων alistarh-herlihy, nuddle και smartpq για ποικίλα αρχικά μεγέθη της ουράς προτεραιότητας και διάφορα ποσοστά των λειτουργιών lookup/insert/deleteMin. Παρατηρούμε ότι η smart δομή επιλέγει κάθε φορά την πιο αποδοτική υλοποίηση.

συνάρτησης διατηρεί ή όχι τον ίδιο τρόπο εκτέλεσης. Πιο συγκεκριμένα, όταν το αποτέλεσμα της συνάρτησης είναι 1, όλα τα νήματα μπορούν να προσπελάσουν απευθείας τη δομή (numa-oblivious alistarh-herlihy), ενώ όταν είναι 2, κάποια νήματα θα λειτουργούν ως clients και άρα θα πρέπει να στείλουν τα αιτήματά τους στους servers οι οποίοι θα τα εκτελέσουν στη δομή. Τέλος, εάν το αποτέλεσμα αντιστοιχεί στη ουδέτερη κλάση, η smart ουρά προτεραιότητας θα διατηρήσει τον τρόπο εκτέλεσης που είχε και πριν. Έτσι, περιορίζονται οι μη αναγκαίες μεταβάσεις που θα μπορούσαν να μειώσουν το throughput



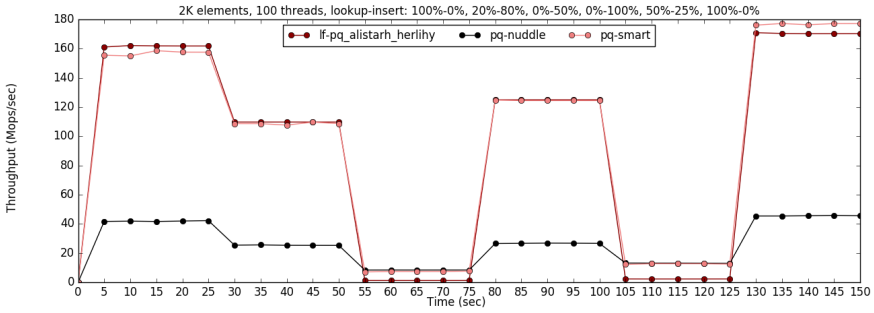
Σχήμα 1.18: Η smartpq μπορεί να μεταβεί από το numa-aware τρόπο εκτέλεσης (Nuddle) στον αντίίστοιχο numa-oblivious, σύμφωνα με την πρόβλεψη του classifier χωρίς να προσθέτει επιπλέον overhead.

της δομής.

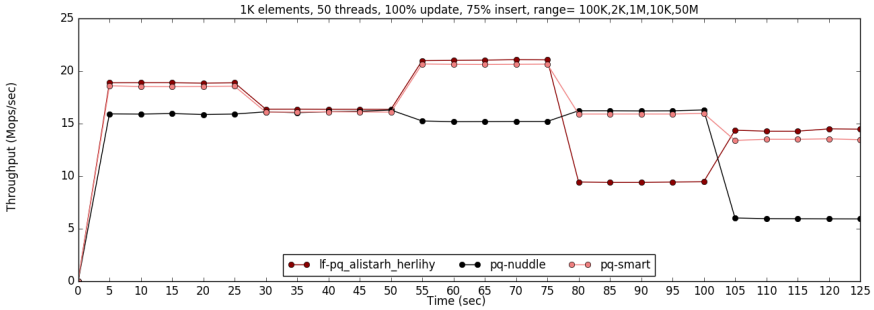
Το βασικό πλεονέκτημα της υλοποίησής μας έγκειται, όπως αναφέραμε και παραπάνω, στο γεγονός ότι οι δύο τρόποι εκτέλεσης , δηλαδή η numa-aware και η numa-oblivious υλοποιήση χρησιμοποιούν την ίδια εσωτερική αναπαράσταση της ουράς προτεραιότητας και τις ίδιες υλοποιήσεις των βασικών λειτουργιών της. Το γεγονός αυτό σημαίνει, ότι, ακόμα και σε περιόδους μετάβασης μεταξύ των δύο υλοποιήσεων, κάποια νήματα-clients μπορούν να εκτελούν τις λειτουργίες τους μέσω των servers, ενώ άλλα μπορούν να προσπελάσουν απευθείας τη δομή, χωρίς να παραβιάζεται η ορθότητα της ουράς. Σε άλλη περίπτωση, εάν οι δύο υλοποιήσεις δεν βασίζονταν στον ίδιο αλγόριθμο, θα έπρεπε να προστεθεί ένα σημείο συγχρονισμού κατά τις μεταβάσεις από τον έναν τρόπο εκτέλεσης στον άλλον, για να εξασφαλιστεί η ορθότητα της παράλληλης εκτέλεσης. Ο συγχρονισμός των νημάτων μεταξύ των δύο καταστάσεων θα μπορούσε να επιτευχθεί μέσω ενός barrier και είναι πολύ πιθανόν να επέβαλε επιπλέον overhead και άρα να μείωνε την απόδοση.

Αξιολογήσαμε πειραματικά τη smart υλοποίησή μας και τη συγκρίναμε με τις υλοποιήσεις nuddle και alistarh-herlihy χρησιμοποιώντας μια σειρά από δυναμικά benchrark, αποτελούμενα από 2 ή παραπάνω περιόδους. Ορίζουμε σαν περίοδο ένα χρονικό διάστημα 25 δευτερολέπτων κατά το οποίο το εύρος των στοιχείων που εισάγονται ή αναζητούνται στην ουρά, ο τύπος και τα ποσοστά των λειτουργιών στη δομή καθώς και ο αριθμός των νημάτων που εκτελούν αυτές τις λειτουργίες παραμένει σταθερός. Μετά το τέλος μιας περιόδου ένα ή περισσότερα από αυτά τα γνωρίσματα μεταβάλλεται. Μετά από κάθε περίοδο, όλα τα νήματα της smart υλοποίησης καλούν τη συνάρτηση-ταξινομητή με παραμέτρους τα νέα γνωρίσματα και ανάλογα με το αποτέλεσμά της καθορίζουν αν θα πρέπει να αλλάξουν κατάσταση εκτέλεσης ή όχι. Ενδεικτικά παρουσιάζουμε στα διαγράμματα 1.19 και 1.20 κάποια από τα αποτελέσματα των πειραμάτων μας, όπου μεταβάλλουμε σε

κάθε περίοδο τα σχετικά ποσοστά των λειτουργιών και το εύρος των στοιχείων αντίστοιχα. Παρατηρούμε ότι σε όλες τις περιόδους, η smart υλοποίησή μας επιλέγει κάθε φορά τον πιο αποδοτικό τρόπο εκτέλεσης και επιτυγχάνει την υψηλότερη δυνατή απόδοση.



Σχήμα 1.19: Εκτελέσαμε τις υλοποιήσεις Nuddle, alistarh-herlihy και smart για 6 περιόδους των 25 δευτερολέπτων. Σε κάθε περίοδο μεταβάλλουμε τα ποσοστά των lookup/insert/deleteMin λειτουργιών. Η smart ουρά προτεραιότητας επιλέγει κάθε φορά τον πιο αποδοτικό τρόπο εκτέλεσης.



Σχήμα 1.20: Οι υλοποιήσεις Nuddle, alistarh-herlihy και smart εκτελέστηκαν για 125 δευτερόλεπτα με το εύρος των στοιχείων που εισάγονται στη δομή να μεταβάλλεται κάθε 25 δευτερόλεπτα.

1.9 Συμπεράσματα και Μελλοντικές Επεκτάσεις

Στο πρώτο μέρος της διπλωματικής αξιολογήσαμε διάφορες υλοποιήσεις παράλληλων δομών δεδομένων που εμπεριέχονται στη βιβλιοθήκη του Ascylib [10] σε ένα σύστημα τύπου numa. Παρατηρήσαμε ότι η απόδοση των υπό εξέταση παράλληλων ουρών προτεραιότητας μειώνεται δραματικά όταν εκτελούνται σε μια NUMA πλατφόρμα, και για αυτό το λόγο επικεντρώσαμε τη μελέτη μας στο συγκεκριμένο τύπο δομών δεδομένων, καθώς χρησιμοποιείται σε ένα ευρύ φάσμα εφαρμογών. Η μείωση της απόδοσης των παράλληλων υλοποιήσεων οφείλεται κατα κύριο λόγο στο numa effect, το οποίο προκύπτει όταν το πλήθος των απομακρυσμένων προσβάσεων μνήμης είναι μεγάλο, και άρα η κίνηση στο

δίκτυο διασύνδεσης γίνεται πιο έντονη αυξάνοντας έτσι το latency των λειτουργιών. Μία από τις βασικές λειτουργίες στις ουρές προτεραιότητας είναι η διαγραφή του στοιχείου με τη μικρότερη ή μεγαλύτερη προτεραιότητα. Η λειτουργία αυτή εξαναγκάζει τα νήματα να ανταγωνίζονται για ένα μικρό αριθμό από cache lines και προκαλεί το numa effect.

Για την επίλυση αυτού του προβλήματος, οι ερευνητές έχουν στραφεί προς το σχεδιασμό numa-aware δομών δεδομένων οι οποίες στοχεύουν να περιορίσουν το numa effect. Οι numa-aware υλοποιήσεις βασίζονται κυρίως σε τεχνικές όπως το combining και το delegation (λειτουργία δια αντιπροσώπου), οι οποίες εκμεταλλεύονται την απόδοση ενός numa node και περιορίζουν την κίνηση στο δίκτυο διασύνδεσης. Στην παρουσα διπλωματική εμβαθύναμε στην τεχνική του delegation, σύμφωνα με την οποία μόνο κάποια από τα νήματα (οι servers) μπορούν να προσπελάσουν τη δομή και εκτελούν λειτουργίες εκ μέρους των υπολοίπων νημάτων (clients). Μελετήσαμε και αξιολογήσαμε πειραματικά τον αλγόριθμο *fast fly-weighted delegation (FFWD)*, μια υλοποίηση του delegation. Σύμφωνα με το FFWD, υπάρχει ένας server που επεξεργάζεται τα αιτήματα των clients και έχει πρόσβαση στη δομή (σε μια ουρά προτερατότητας στην περίπτωσή μας), και συνεπώς δεν υπάρχει ανάγκη συγχρονισμού στη δομή. Οι clients επικοινωνούν με τον server μέσω ενός πρωτοκόλλου επικοινωνίας το οποίο χρησιμοποιεί συγκεκριμένο αριθμό από cache lines και άρα καταφέρνει να περιορίσει το overhead της επικοινωνίας μεταξύ clients-server. Παρόλα αυτά, η απόδοση του FFWD περιορίζεται από το ρυθμό με τον οποίο ο μοναδικός server επεξεργάζεται τα αιτήματα των clients και τα εκτελεί στη δομή.

Για να ξεπεράσουμε αυτό τον περιορισμό επεκτείναμε το FFWD προσθέτοντας περισσότερους servers. Σχεδιάσαμε και υλοποιήσαμε τον αλγόριθμο NUDDLE (Numa Node Delegation), ο οποίος μπορεί να μετατρέψει μια numa-oblivious παράλληλη δομή δεδομένων στην αντίστοιχη numa-aware. Οι servers τοποθετούνται στον ίδιο numa node και άρα η δομή παραμένει στην ιεραρχία μνήμης του συγκεκριμένου numa node, μειώνοντας έτσι τον αριθμό των προσβάσεων σε απομακρυσμένη μνήμη. Επιπλέον, εφόσον πολλαπλοί servers έχουν πρόσβαση στη δομή αυξάνεται ο παραλληλισμός των λειτουργιών. Σχεδιάσαμε, λοιπόν, μια numa-aware ουρά προτεραιότητας χρησιμοποιώντας την τεχνική NUDDLE και μια παράλληλη υλοποίηση ουράς προτεραιότητας βασισμένη σε skiplist που προτείνεται στο [2]. Τα πειράματά μας έδειξαν ότι σε περιπτώσεις υψηλού ανταγωνισμού μεταξύ των νημάτων, το NUDDLE αποδεικνύεται καλύτερο από τις numa-oblivious παράλληλες υλοποιήσεις, αλλά και πάλι η απόδοσή του περιορίζεται από τον αριθμό των servers. Παρατηρήσαμε, λοιπόν, ότι δεν υπάρχει μια υλοποίηση που να είναι η καλυτερη από τις υπόλοιπες κάτω από όλες τις συνθήκες, καθώς κάποιες φορές υπάρχει μεγαλύτερη ανάγκη για διατήρηση των δεδομένων σε μια συγκεκριμένη ιεραρχία μνήμης και άρα πρέπει να χρησιμοποιηθεί μια numa-aware τεχνική ενώ απαιτείται άλλες φορές μια numa-oblivious υλοποίηση καθώς υπάρχει ανάγκη για περισσότερο παραλληλισμό.

Τα πειράματά μας έδειξαν ότι παράγοντες όπως το μέγεθος της δομής, ο αριθμός των νημάτων, το εύρος των στοιχείων της δομής, το είδος και το πλήθος των λειτουργιών επηρεάζουν τους υπο εξέταση αλγορίθμους με έναν τρόπο που είναι δύσκολο να μοντελοποιηθεί και άρα δεν είναι εύκολο να προβλέψουμε ποια είναι κάθε φορά η πιο αποδοτική υλοποίηση. Έτσι, χρησιμοποιήσαμε τεχνικές μηχανικής μάθησης και μοντελοποιήσαμε το πρόβλημα της επιλογής του πιο αποδοτικού αλγορίθμου ως ένα πρόβλημα ταξινόμη-

σης με είσοδο τους παραπάνω παράγοντες (features) και έξοδο μία από τις εξής τρεις κλάσσεις: numa-aware, numa-oblivious, ουδέτερη. Η numa-aware κλαση δηλώνει ότι ο αλγόριθμος NUDDLE είναι προτιμητέος στη συγκεκριμένη περίπτωση, ενώ η κλάση numa-oblivious συμβολίζει τον αλγόριθμο alistarh-herlihy. Στην περίπτωση της ουδέτερης κλάσης η συμπεριφορά των δύο αλγορίθμων είναι παρόμοια. Έτσι, εκπειδεύσαμε ένα decision tree ταξινομητή για το συγκεκριμένο πρόβλημα και καταλήξαμε σε ένα μοντέλο που προβέπει τη σωστή κλάση με ακρίβεια 93%. Στη συνέχεια, υλοποιήσαμε μια smart ουρά προτεραιότητας, η οποία χρησιμοποιεί το παραπάνω μοντελο για να προσδιορίσει αν θα πρέπει, δοθέντων συνθηκών εκτέλεσης, να λειτουργήσει είτε σύμφωνα με τον αλγόριθμο NUDDLE είτε σύμφωνα με τον αλγόριθμο alistarh-herlihy. Τέλος, προσομοιώσαμε ένα δυναμικά μεταβαλλόμενο περιβάλλον, αλλάζοντας ανά τακτά χρονικά διαστήματα τις τιμές των παραπάνω features. Καθώς οι συνθήκες εκτέλεσης μεταβάλλονται, μια δομή θα πρέπει να προσαρμοστεί σε αυτές τις αλλαγές ώστε να διατηρήσει υψηλή απόδοση. Η smart υλοποίησή μας, μπορεί να μεταβεί από το numa-aware στο numa-oblivious τρόπο εκτέλεσης χωρίς να επιβάλει επιπλέον overhead και έτσι επιτυγχάνει την υψηλότερη δυνατή απόδοση κάτω από όλες τις συνθήκες

Η smart ουρά προτεραιότητας θα μπορούσε να επεκταθεί ώστε, κατά την εκτέλεσή της, να πραγματοποιείται εξαγωγή πληροφοριών για τους παράγοντες που επηρρεάζουν την απόδοσή της. Στα πειράματά μας, γνωρίζαμε εξαρχής πότε και πώς άλλαζαν οι συγκεκριμένοι παράγοντες, και μόνο το μέγεθος της δομής ήταν άγνωστο και έπρεπε να βρεθεί κατά τη διάρκεια της εκτέλεσης. Ως μελλοντική επέκταση θα μπορούσαμε να εξάγουμε με κάποιο στατιστικό τρόπο και τα υπόλοιπα χαρακρηριστικά όσο ο αλγόριθμος εκτελείται. Μια πιθανή προσέγγιση θα ήταν ένα νήμα να κατέγραφε περιοδικά πληροφορίες σχετικά με τα συγκεκριμένα χαρακτηριστικά, και με βάση το decision tree μοντέλο να ενημέρωνε τα υπόλοιπα νήματα εάν υπήρχε η ανάγκη για αλλαγή τρόπου εκτέλεσης. Μια άλλη λύση θα ήταν να χρησιμοποιούσαμε επιπλέον πεδία στη δομή, όπου κάθε νήμα θα κατέγραφε ατομικά κάποια στατιστικά στοιχεία σχετικά με τις λειτουργίες του ανά τακτά χρονικά διαστήματα.

Πιστεύουμε ότι η μεθοδολογία που προτείνουμε στην παρούσα διπλωματική αποτελεί ένα βήμα προς την κατεύθυνση του σχεδιασμού δυναμικών παραάλληλων δομών δεδομένων. Παρόλο που επικεντρωθήκαμε στις ουρές προτεραιότητας, οι μεθοδολογίες που παρουσιάσαμε μπορούν να χρησιμοποιηθούν και σε άλλες δομές δεδομένων όπως οι στοιβες, οι ουρές FIFO κ.α., που χαρακτηρίζονται από χαμηλά επίπεδα παραλληλισμού και επηρεάζονται έντονα από το numa effect. Επιπλέον, όσον αφορά το κομμάτι της μηχανικής μάθησης, υπάρχουν πολλά περιθώρια έρευνας για το σχεδιασμό του καταλληλότερου μοντέλου, όπως η δοκιμή διαφόρων ταξινομητών και τεχνικών βελτίωσης ακρίβειας, τεχνικών online μάθησης, καθώς και η επανεξέταση των features του προβλήματος. Η μηχανική μάθηση παρέχει πολύ χρήσιμα εργαλεία για το σχεδιασμό αποδοτικών παράλληλων δομών δεδομένων, και άρα κλιμακώσιμων παράλληλων αλγορίθμων.

Chapter 2

Introduction

2.1 Overview

We are surrounded by multicore systems. From enormous supercomputers with large interconnection networks to distributed systems and personal computers, the number of cores is being increased rapidly, resulting to computationally powerful machines. Servers with large numbers of cores and modern architectures are used to process huge amounts of data on a daily basis. As a result, there is an increasing need for more efficient and scalable applications that can exploit the given parallelism. Concurrent applications are used everywhere, from medicine and physics to economics and banking systems, requiring correctness and direct responsiveness. Therefore, the right and efficient design of these applications is a crucial issue that programmers face.

Despite the fact that the evolution in hardware offers the ability for fast and efficient applications, there are some challenges and constraints that programmers should have in mind when designing multithreaded applications. Sequential algorithms should be adapted and redesigned in order to be able to run on multicore systems. Synchronization, communication between threads and memory management are only some of the problems that occur in parallel programming. Furthermore, the scalability of a concurrent algorithm is constrained by Amdahl's law, according to which, the speedup of an application is limited by the percentage of the program that cannot be parallelized and must be executed sequentially. This percentage relates mostly to inter-thread communication and shared-memory issues, something that becomes clear with concurrent data structures.

Modern applications are based on high-performance concurrent data structures. Designing an ideal data structure that scales well when the number of threads grows, and performs the best regardless of the machine's specific architecture remains an unsolved problem. The main factor of the structure's implementation is how to ensure fast, secure and efficient access to the shared data between the threads. Furthermore, the majority of modern multicore servers are non-uniform memory access (NUMA) machines, and in order to fully exploit these servers, NUMA-aware data structures are necessary.

2.2 Parallel Architectures

Figure 2.1 presents the Flynn's Taxonomy of Computer Architectures.

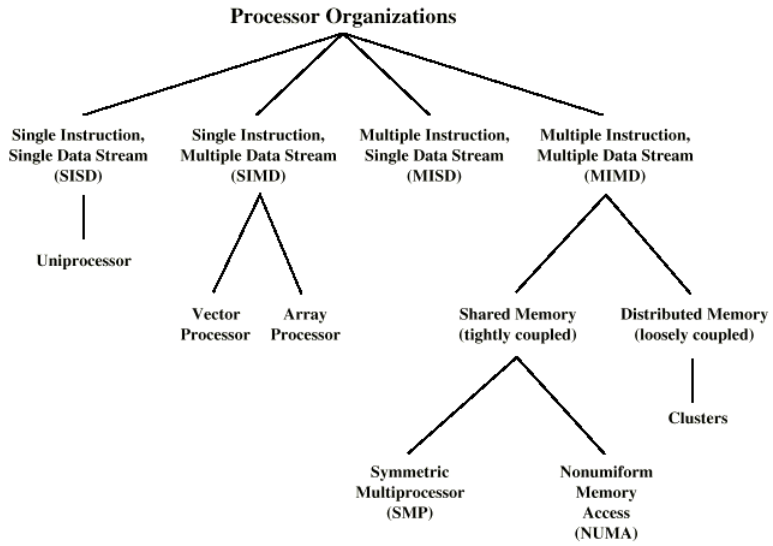


Figure 2.1: A taxonomy of parallel processor architectures

At this thesis, we focus on the Multiple Instruction Multiple Data (MIMD) Stream, which is the most common organization in today's systems. As shown in figure 2.2, MIMD is used to achieve parallelism: a number of processors can function independently, executing different operations on different data, building a multiprocessor system. There are 3 criteria on which performance of a multiprocessor system can be judged: **scalability**, **latency** and **bandwidth**. Scalability is the ability of a system to demonstrate a proportional increase in parallel speedup with the addition of more processors. Latency is the time taken in sending a message from node A to node B, while bandwidth is the amount of data that can be communicated per unit of time. Thus, the goal of a multiprocessor system is to be highly scalable with low latency and high bandwidth.

MIMD machines can either have shared, distributed or hybrid architecture. In this section we are focusing on the characteristics of each type of parallel architecture, and the way they satisfy the aforementioned criteria.

2.2.1 Shared Memory Architecture

In *shared memory architectures*, all processors share the same "global" memory and have the same physical memory address space. They may operate independently but they share the same resources. Processors can access the data stored in memory through the shared bus that connects them, which also handles requests to I/O. Each processor has a private local cache where recent copies of data are stored. Tasks can be executed con-

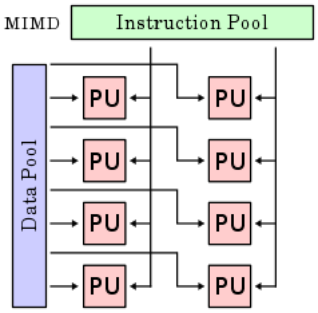


Figure 2.2: MIMD layout

currently by accessing the shared data with loads/stores and making copies of them at the processor's caches, creating the need for a Memory Coherence Protocol.

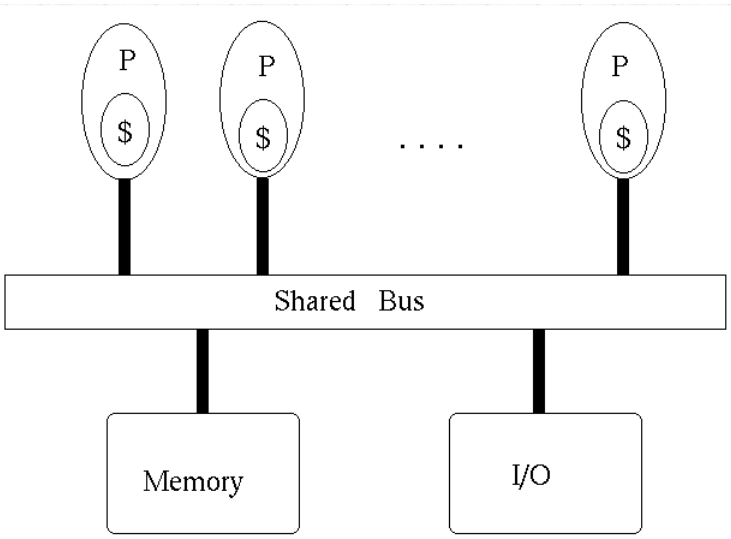


Figure 2.3: Shared memory architecture

Shared memory systems may use Uniform Memory Access (UMA) or Non-Uniform Memory Access (NUMA), into which we will delve at the next section. Briefly, if the amount of time to access the global memory is the same for all processors then the memory organization is characterized as uniform, and is most commonly represented today by Symmetric Multiprocessor (SMP) machines. On the other hand, if some global memory accesses are quicker than others, the memory system is characterized as non-uniform.

The ease of access to the global variables, with loads and stores, makes Shared Memory Architectures friendly to programmers. In parallel programming, though, concurrent accesses to the same data result into race conditions and synchronization mechanisms such as locks, monitors, semaphores, atomic variables etc are demanded. Conventional multicore processors directly support shared memory, which many parallel programming lan-

guages and libraries, such as Cilk, OpenMP and Threading Building Blocks, are designed to exploit.

Shared memory systems offer fast and uniform data sharing between the tasks and provide a user-friendly programming perspective to memory, due to the shared bus. Nevertheless, the bus is the one causing the lack of scalability between the memory and the CPU. In these systems, adding more CPUs can increase traffic on the shared bus, due to multiple requests to the memory and the cache coherence protocol.

2.2.2 Distributed Memory Architecture

In *distributed memory architectures*, each processor has its own private cache and local memory, and it cannot directly access another processor's main memory. Memory addresses in one processor do not map to another processor, and therefore there is no concept of global address space across all processors. Processors are connected with Interconnection Networks, which vary according to the size and the needs of the system. All communication and synchronization between processors is accomplished via messages passed through the Interconnection Network, and this is why this type of architecture is called *message passing architecture*. We can find distributed architectures at clusters, that are groups of computers (*nodes*) connected with an interconnection network.

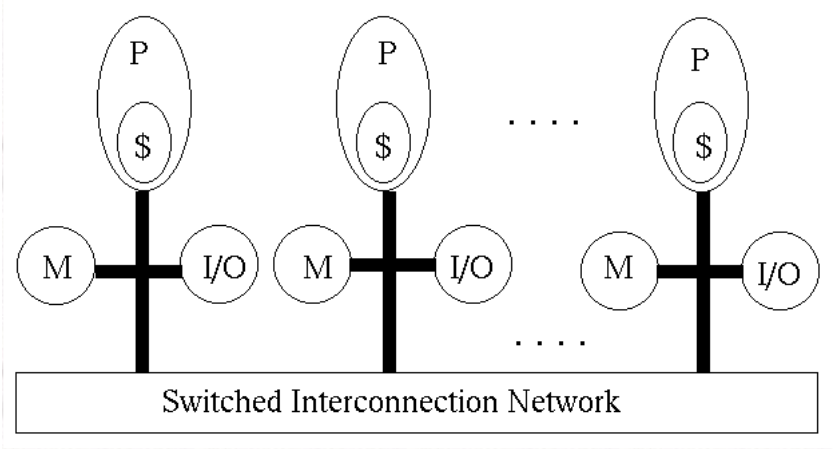


Figure 2.4: Distributed memory architecture

Given that each processor has its own local memory, it operates independently. The updates to its local memory have no effect on the memory of other processors. Hence, the concepts of cache coherency as well as of race condition do not apply. When a processor needs access to data in another processor, it is usually task of the programmer to explicitly define how and when data is communicated. Synchronization between tasks is also the programmer's responsibility. Parallel Programming in Distributed Memory Systems is based on message passing between the processors using libraries such as MPI and MPIV.

One of the greatest advantages of distributed memory systems is the fact that the memory is scalable with the number of processors. Increasing the number of processors pro-

portionately increases the size of memory. Each processor can access its own memory without interference and without the overhead incurred by trying to maintain cache coherency. On the other hand, it is the programmer's duty to manage all details about the inter-processors communication and the exchange of data between the processors' local memories. If the time needed for communicating between the processors exceeds the actual time of execution, then the performance can be degraded. As a result, it depends on the needs of the specific problem whether the implementation would be better in a shared or in a distributed environment.

2.2.3 Hybrid Memory Architecture

Hybrid memory architectures combine the two previous types. The components of this architecture are usually cache-coherent Symmetric Multiproccessors (SMP) machines, often called as nodes. Processors on a given SMP can address that machine's memory as global. The distributed memory component is the networking of multiple SMPs. SMPs know only about their own memory - not the memory on another SMP. Therefore, interconnection networks are required to move data from one SMP to another.

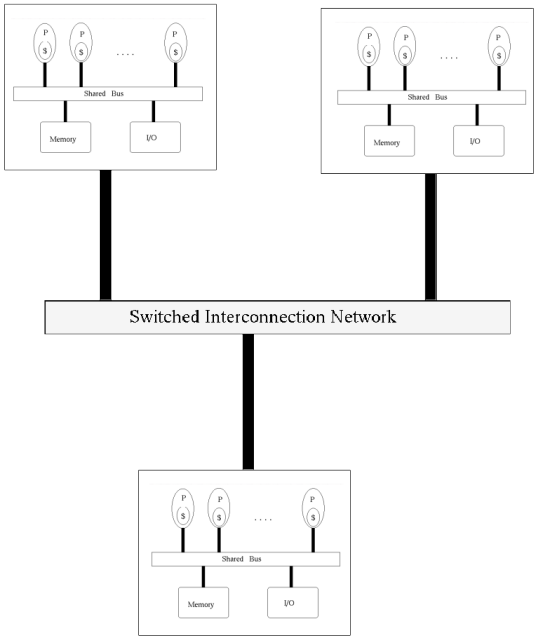


Figure 2.5: Hybrid memory architecture

Hybrid Architectures combine advantages from both Shared and Distributed Mem-

ory Systems. Current trends seem to indicate that this type of memory architecture will continue to prevail and increase at the high end of computing for the foreseeable future.

2.3 Numa Architecture

In this section we are focusing on Non-Uniform Memory Access Architectures. We explain the main reasons that led to the development of NUMA machines, the characteristics of this particular type of architecture, the programming challenges introduced by this type of architectures and we also present methods to overcome them.

2.3.1 From UMA to NUMA

Figure 2.6 is a simplified representation of an UMA architecture. All processors can use the memory-cpu bus and access all regions of the shared memory within an equal amount of time. This organization is used in symmetric multiprocessors machines, the throughput of which is limited by the bandwidth of the bus. Bus traffic limits to between 16 and 32 processors and as a result SMP machines suffer from low scalability. NUMA alleviates the bottlenecks that the memory-cpu bus imposes by limiting the number of CPUs on a single *node*, and connecting the various nodes using a high speed interconnection network.

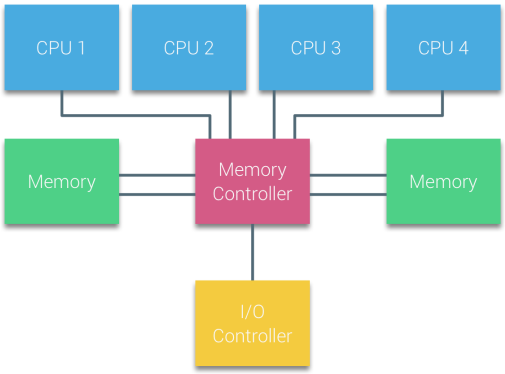


Figure 2.6: Uniform Memory Access architecture

A NUMA system classifies memory into NUMA nodes (what Solaris calls locality groups). Each NUMA node (or **socket**) has its own portion of main memory and its own memory controller. Figure 2.7 presents the layout of a NUMA architecture with four *numa nodes*. All memory available in one node has the same access characteristics for a particular processor. Nodes have an affinity to processors and to devices. These are the devices that can use memory on a NUMA node with the best performance since they are locally attached. Memory is called *node local* if it was allocated from most suitable, for the processor, NUMA node. A memory access from one socket to memory of another

has additional latency overhead to accessing local memory - it requires the traversal of the memory interconnect first. On the other hand, accesses from a single processor to local memory not only have lower latency compared to remote memory accesses, but also do not cause contention on the interconnect and the remote memory controllers.

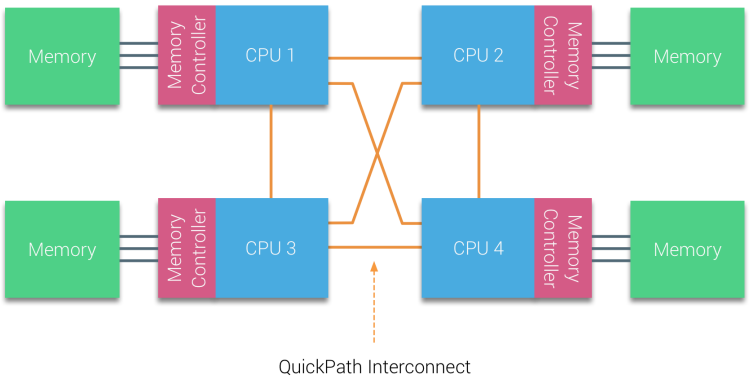


Figure 2.7: Non Uniform Memory Access architecture

NUMA systems that maintain cache coherence are referred to as **ccNUMA** machines. Since few applications still exist for non-cache coherent NUMA machines, the terms NUMA and ccNUMA are used interchangeably. In order to maintain cache coherency and to serve the processors' requests for data, each node maintains directory of location of portions of memory and cache status. When a processor makes a memory request, at first the different levels of cache hierarchy (L1,L2,L3) are examined, then the local-to-node main memory and finally the remote memory. When the requested data are located in remote memory, data need to be transferred through the network at the local cache. This process is automatic and transparent in NUMA machines, and this is what differentiates them from clusters: The programmer is not responsible neither for the communication between the processors nor for the maintenance of cache coherency.

The maximum available bandwidth of a NUMA machine is the sum of the peak bandwidth of each memory controller. On modern NUMA machines that only send cache coherency messages when required, the maximum bandwidth can be achieved when all cores access their local memory. When all cores access their local node, no cache coherency message is sent and the interconnect links are not used - thus avoiding latencies or bandwidth limitations due to inter-node communications.

2.3.2 Challenges introduced by NUMA

As a result, it becomes clear that in order to maximize the usage of a NUMA machine, a developer has to minimize the number of remote memory accesses and to make sure that the load is balanced between the memory controllers. However, in many applications this is not possible. In such cases, the remote memory accesses and the contention of memory controllers' bandwidth lead to degradation of performance.

This characteristic of NUMA architectures that causes low performance and often demands re-designing of concurrent algorithms is called **numa effect**. When data are placed in many numa nodes, the memory requests increase the traffic at the interconnection network and the need for cache coherence makes the situation worse. This leads to a kind of ”ping-pong” of the cache lines between the numa nodes, increases the latency and reduces the bandwidth.

As mentioned at [21], there are some well-known ways to reduce the excess of numa effect such as:

- **Thread placement:** The first technique refers to placing the threads close to the data they access. This is known as thread affinity and may lead to sufficient optimization but also can cause load imbalancing at CPUs.

- **Numa node-local memory allocation:** One common technique is to allocate data at the numa node from where they will be accessed the most, in order to eliminate remote memory accesses. This implies that threads are going to be attached to specific cores and they are not migrating between sockets. This technique works well when data are partitioned at different numa nodes and every access to a remote memory area is actually *delegated* to a thread at this area.

- **Memory migration:** This technique refers to migration of the data to the node that uses them the most. It is a dynamic method which means that data can be migrated multiple times during the execution of the program. However, memory migration imposes an additional cost, and as a result it is best to be used at cases where data are accessed by a particular numa node for a long period of time. Memory migration is unlikely to work well on data that are frequently accessed from different nodes.

- **Memory interleaving:** This method is based on splitting the memory randomly to the various memory controllers and therefore to the various numa nodes, in order to reduce the contention at memory controllers and balance the workload. This can be implemented either statically or dynamically using memory migration.

- **Memory replication:** The last method proposes the replication data on multiple nodes making sure that threads access the replica located on their local node. In that case, there is a gain in memory locality but an extra cost is invoked due to the creation of replicas and the maintenance of consistency between them.

The aforementioned techniques do not have the same effect to all applications. As a result, in order to actually optimize a concurrent application, the programmer should be aware of its specific memory patterns and data accesses. However, it is not always easy to predict the behavior of a concurrent application and as a result, one should try the most appropriate solutions, and deal with the trade offs that may come up.

2.4 Parallel Programming

In this section we present the main principles of parallel programming and we make a brief reference to some important *data structures* and *synchronization techniques*.

2.4.1 Amdahl's Law

Ideally, the performance of any application executed on a N-core machine would have N times higher performance than executed on a single-core machine. In fact this is not happening, because no matter how effective a concurrent implementation can be, one cannot simply eliminate the costs of inter-processor communication and coordination. The most known formula that describes this phenomenon is called Amdahl's Law: the extent to which we can speed up any complex job is limited by how much of the job must be executed sequentially. More specifically, let the *speedup S* of a parallel job be the ratio of the time it takes one processor to complete the job versus the time it takes N concurrent processors to complete the same job. Furthermore, *p* is the fraction of the job that can be executed in parallel, and *N* is the number of the processors that will execute the job concurrently. Then, let us assume that it takes (normalized) time 1 for a single processor to complete the job. With N concurrent processors, the parallel part takes time p/N and the sequential part takes time 1 − p. Overall, the parallelized computation takes time:

$$T = 1 - p + \frac{p}{N}$$

According to Amdahl, the speedup which is, the ratio between the sequential (single-processor) time and the parallel time, is:

$$S = \frac{1}{1 - p + \frac{p}{N}}$$

We can easily assume that when $N \to \infty, S \to \frac{1}{1-p}$.

This formula implies that the the total speedup of a parallel job is bounded by the sequential part of the program and using more processors does not increase the speedup in all cases. This sequential part is due to data dependencies and synchronization needs. Therefore, in order to increase speedup, we need to parallelize more sequential code and re-design our well-known sequential algorithms.

2.4.2 Memory Coherence

Modern memory architectures are based on caches which maintain copies of the main memory, that the processors have recently used. As a result, it is much easier and quicker for a thread to access the data that have already been cached. Although caches reduce the average data access time and the bandwidth demands at the shared interconnection bus, the existence of multiple copies of the same memory location can cause many problems, especially at multicore systems. In these systems, there are two or more threads working

at the same time, and thus it is possible that they simultaneously access the same memory location. Provided none of them changes the data in this location, they can share it indefinitely and cache it as they please. But as soon as one updates the location, the other might work on an out-of-date copy that resides in their local cache. Consequently, some scheme is required to notify all the processing elements of changes to shared values; such a scheme is known as a **memory coherence protocol**, and if such a protocol is employed the system is said to have a coherent memory.

Cache coherence protocols are either bus-snooping protocols or directory schemes. One of the most commonly used protocol is **MESI** presented at figure 2.8.

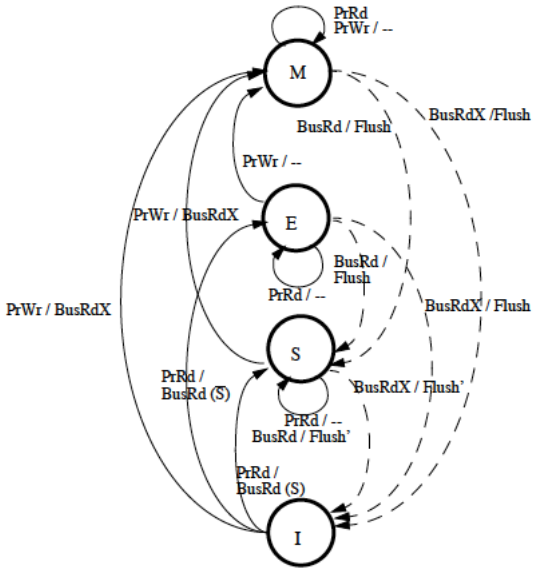


Figure 2.8: MESI protocol

The MESI protocol defines 4 possible situations for a cache line:

- **Modified:** Only this cache line contains the latest version of data which do not match main memory. The cache is required to write the data back to main memory at some time in the future, before any read to them.
- **Exclusive:** Only this particular cache line contains the data which are the latest version and matches the same memory. It may be changed to the Shared state at any time, in response to a read request from another processor. Alternatively, it may be changed to the Modified state when the owner-processor writes to it.
- **Shared:** This cache line matches main memory and it is the latest version of data. The last version of the data may also be in another processor cache. It can be changed to Modified if the owner-processor writes to it, or to Invalid if another processor sends a write request.
- **Invalid:** The cache line contains an outdated version of the data.

Extensions of MESI protocol are MOESI and MESIF protocols.

2.4.3 Memory Consistency

While coherence guarantees a single order of all writes to the same location, memory consistency specifies the ordering of loads and stores to different memory locations. A memory consistency model is a set of rules which specify when a written value by one thread can be read by another thread. Without these rules it is not possible to write a valid parallel program, that has the same output with the corresponding sequential execution. The memory consistency model also affects which programmer/compiler and hardware optimization is legal. In a parallel program, unlike to a single-threaded execution, multiple correct behaviours are usually allowed. Memory consistency protocols can either be **sequential** or **relaxed**.

The most intuitive model is the sequential model, which ensures the **total ordering** of reads and writes. This means all memory operations must complete according to how they were issued. The easiest way to implement this model is to serialize all memory accesses. However, memory reads and writes can take long time, thus this model can easily become a bottleneck at the system. The sequential model is the easiest to be implemented in both hardware and software level. For software, the programmer does not have to worry about out of order instruction and each instruction will be executed with the order which the program implies. For hardware, all instruction can be simply put into a FIFO queue.

On the other hand, relaxed consistency models may violate certain orderings, but memory utilization can be greatly improved. Different models of relaxed consistency allow different violations, and consequently different results. Relaxed memory models are better from sequential at multiprocessor systems and allow more compiler optimization. Total Store Ordering, Processor Consistency and Partial Consistency are three of the most known relaxed consistency models. These need extra effort from both the hardware designer and programmer to ensure the correctness of the program. Extra synchronization mechanisms such as locks and memory barriers have to be used to help ensure correctness.

2.4.4 Concurrent Objects

This subsection presents the main properties that characterize concurrent objects, i.e., objects consisting of operations acting on shared data that may be executed concurrently by multiple processes. These properties refer to the objects' **Correctness** (or safety) and **Progress** (or liveness).

2.4.4.1 Correctness

Even though it is simple to define the correctness of a sequential object, in multi-threaded programs where the execution paths of many threads are interleaving, the notion of correctness is hard to be captured and even harder to be maintained. An easy way to do so is to map any concurrent object to its corresponding sequential one and find out the

way they are related. We will try to explain the three main conditions of correctness which are: *Quiescent Consistency*, *Sequential Consistency* and *Linearizability*.

- **Quiescent Consistency:** An execution of a concurrent program is quiescently consistent if its method calls can be correctly arranged retaining the mutual order of calls separated by quiescence, a period of time where no method is being called by any thread. It is a relatively relaxed consistency model, that imposes less restrictions in implementing concurrent programs. It is appropriate for systems that require high performance and lower response time at the cost of placing relatively weak constraints on object behaviour.

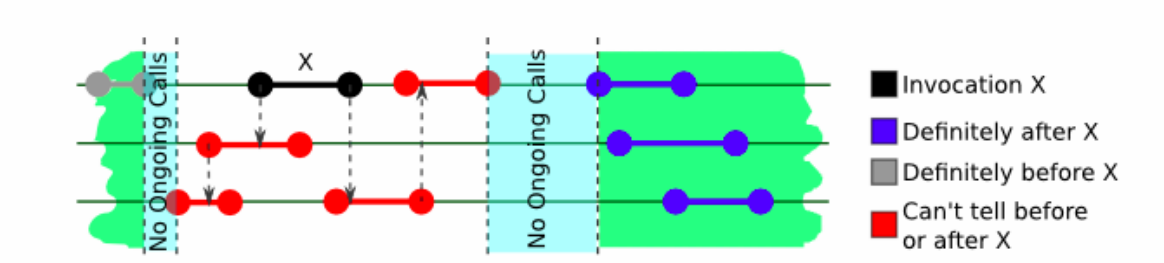


Figure 2.9: An example of Quiescent Consistency

Figure 2.9 shows an example of three threads executing their operations. The times of execution are interleaved with times of quiescence. This model of correctness guarantees that sets of operations separated by times of quiescence will maintain their order. Although, there is no guarantee about the order of operations even by a singe thread inside each set.

- **Sequential Consistency:** An execution is sequentially consistent if the method calls can be correctly arranged retaining the mutual order of method calls in each thread. There are not any guarantees or rules for the order of method calls between the different threads, as shown in figure 2.10. We are aware only for the order of calls executed by each thread (e.g. thread X) but not for the total ordering between the threads.

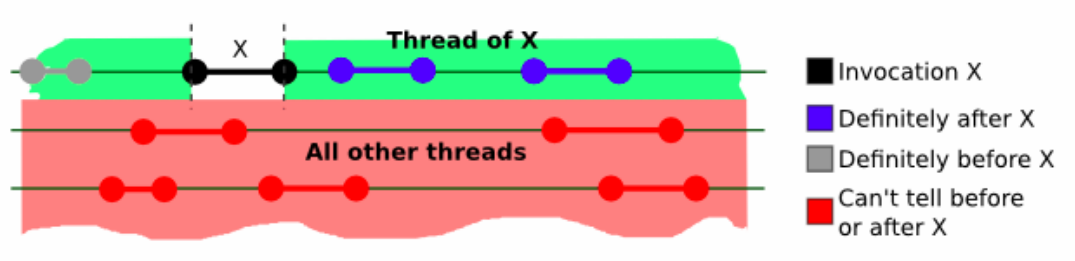


Figure 2.10: An example of Sequential Consistency

More specifically, sequential consistency refers to the maintenance of *program order* i.e the order in which a single thread issues its method calls. Due to compiler optimization, in most modern multiprocessor systems, memory reads and writes are not sequentially consistent and may be reordered. As a result, when programmers need sequential consistency they have to declare it, e.g. using barriers or fences.

- **Linearizability:** The third condition of correctness is the most strict of them: an execution is linearizable (or strongly consistent) if its method calls can be correctly arranged retaining the mutual order of calls that do not overlap in time.

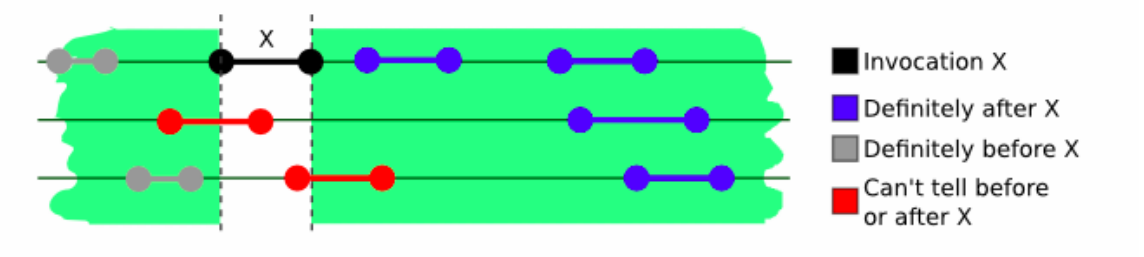


Figure 2.11: A Linearizability example

Therefore, as shown in figure 2.11 if two or more calls overlap (like invocation X overlaps with the red ones), then those may have happened in any order, but if a method returns than we are sure it has already taken effect. Two linearizable executions combined also form a linearizable execution, which is very important for the creation of correct and safe concurrent algorithms. In order to show that an implementation is linearizable, we are using **linearization points**. These can be either critical sections for lock-based applications or otherwise a point at which the results of a method become visible to other methods regardless of the threads calling them.

2.4.4.2 Progress

Progress properties describe conditions under which the calls and interactions with a concurrent object will finally terminate in an execution. The most important ones are wait-freedom, lock-freedom, obstruction-freedom for non-blocking implementation and starvation-freedom and deadlock-freedom for blocking implementation.

- **non - (blocking):** An implementation is called blocking when a delay from a thread can prevent other threads from making progress, and non-blocking when this cannot happen. For example, in lock based implementations, when a thread holding a lock delays, because of a programming bug, a received signal or simply because of a cache miss or a page fault, the other threads cannot take this lock and the execution discontinues. When designing a lock-based application, the programmer should have in mind this scenario.

- **starvation-free:** In general, starvation is called the case when a process is perpetually denied necessary resources to process its work. In parallel implementations, one of the main reasons of starvation is mutual exclusion between threads regarding the same critical section. Consequently, a method is starvation-free in a blocking implementation when the following occurs: as long as one thread is in the critical section, then some other thread that wants to enter in the critical section will eventually succeed.

- **deadlock-free:** The existence of deadlocks is one of the main problems in parallel programming. A set of threads is deadlocked when each thread in the set is blocked awaiting an event that can only be triggered by another blocked thread in the set. Hastily designed lock-based applications apt to deadlocks. A method is actually deadlock-free when for all critical sections, if two or more processes are trying to enter them, one of them will eventually succeed.

- **wait-free:** Wait-freedom is the strongest non-blocking guarantee of progress, combining guaranteed system-wide throughput with starvation-freedom. An algorithm is wait-free if it guarantees that every call finishes its execution in a finite number of steps. Wait-freedom guarantees that the number of steps is finite, but in practice it may be extremely large and depends on the number of active threads. If there is a bound on the number of steps a method call can take, then it is called *bounded wait-free*. This bound may depend on the number of threads and it is more useful at real-time systems. Finally, when this bound is independent from the number of threads, the method is called *wait-free population oblivious*.

- **lock-free:** A method is lock-free if it guarantees that infinitely often some thread calling this method finishes in a finite number of steps. As a result, an algorithm is lock-free if, when the threads run for a sufficiently long time, at least one of them makes progress. For instance, if N processors are trying to execute an operation, some of the N processes will succeed in finishing the operation in a finite number of steps and others might fail and retry on failure. All wait-free algorithms are lock-free but not vice versa, as lock-freedom may allow one or more threads to starve but guarantees system-wide throughput and progress.

- **obstruction-free:** An algorithm is obstruction-free if at any point, a single thread executed in isolation (with all obstructing threads suspended) will complete its operation in a bounded number of steps. It is not hard to show that lock-freedom is a stronger condition that obstruction-freedom: given a lock-free implementation, if we can keep some single process running forever in isolation, we get an infinite execution with only finitely many completed operations. Therefore, we have the following hierarchy: wait-free > lock-free > obstruction-free.

In order to choose the appropriate progress priority, one should take into account both the application's needs and the characteristics of the underlying platform the application is running on. The wait-free and lock-free conditions guarantee that the execution will make

progress no matter how the threads are scheduled. On the other hand, starvation-free, deadlock-free and obstruction-free conditions are *dependent*: progress occurs only if the underlying platform (i.e. the operating system) provides certain guarantees. Given these guarantees, dependent conditions can lead to simpler and more efficient implementation.

2.4.5 Synchronization Mechanisms

In multithreaded applications where the data are accessed by multiple threads, the need for synchronization between them is more than obvious. There are a lot of synchronization techniques a programmer can use to protect critical sections being accessed from more than one threads at a time. These techniques affect both the correctness and the progress of the concurrent object, and as a result they should be selected keeping in mind not only the concurrent algorithm but also the characteristics of the platform it is executed on. In this subsection we focus on the three main categories of synchronization mechanisms: *mutual exclusion*, *atomic operations* and *transactional memory*.

- **Mutual Exclusion:** This technique is used for designing blocking implementations and it guarantees that only one thread will enter the critical section. These kind of implementations follow the pattern shown in figure 2.12. The challenge in working with blocking applications is to efficiently design entry and exit code in order to ensure mutual exclusion and avoid deadlock and starvation effects.

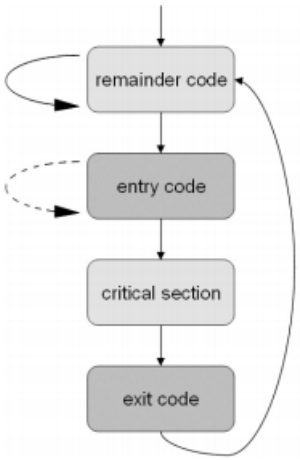


Figure 2.12: mutual exclusion pattern

Mutual exclusion can be implemented with mutexes, semaphores and locks, e.g. variables whose values allow or deny access to critical sections. Many lock algorithms have been proposed, e.g. Peterson's Lock for 2-threads implementations, the Filters lock for multiple threads, Lamport's Algorithm etc.

Today's lock-based implementations use either *coarse* or *fine grained* locking. In fine grained locking, more locks are being used each one protecting a smaller portion of shared data. On the other hand, in coarse grained locking, there is usually

one main lock protecting the data leading to an easier implementation. However, coarse-grained algorithms limit concurrency and suffer from scalability issues due to MESI protocol as the same lock variable is shared by a large set of threads which are sending continuously read and write requests on the corresponding cache line used for that variable .

- **Atomic Operations:** In order to avoid the constraints generated by locks (deadlocks, starvation, performance degradation due to MESI protocol), many concurrent implementations use atomic operations for thread synchronization. As these implementations avoid the blocking effect of lock-based algorithms (e.g. when the thread which holds a specific lock is suspended, and thus other threads are blocked until the suspended thread resumes), they are known as *non-blocking* implementations. An example of non-blocking implementation is shown at figure 2.13.

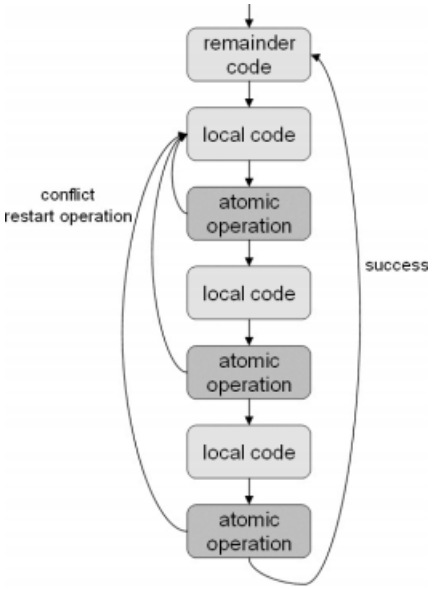


Figure 2.13: Code pattern for a non-blocking implementation

Atomic instructions perform an operation on one or more memory locations atomically, i.e. other threads see them as happening instantaneously. An atomic operation either succeeds or fails in its entirety, regardless of what instructions are being executed by other processors. They are indivisible: other concurrent activities do not interfere with their execution. Fundamental examples of atomic operations are: atomic increment, atomic exchange register and memory location, and compare and swap. More specifically, Compare And Swap (CAS), compares the content of a memory location with a given value and if they are equal, then changes the content to a given new value, otherwise it returns. There are many more atomic instructions and even atomic variables in many programming languages to ensure synchronization between different threads without using locks.

Because atomic instructions can be used to access and operate on shared data without the need to acquire and release a lock, they are quicker, avoid deadlocks and they can allow greater levels of parallelism. They are used to create efficient, scalable lock-free parallel applications especially for concurrent data structures as we will see at the next subsection. However, they perform a limited set of operations, therefore in order to create a totally non-blocking application using only atomic instructions can be a challenging task.

- **Transactional Memory (TM):** Despite the fact that atomic operations can offer a solution at the problems caused by locking, sometimes they have a high overhead. Furthermore, most of them operate on a single word, leading to complex and unstructured algorithms. Another way of creating non-blocking concurrent applications is by using transactional memory. A transaction is a set of operations executed atomically by a single thread. When a thread executes a transaction, this can either **commit** or **abort**. If no conflicts are met, then the transaction commits the changes done to the shared data, otherwise these changes are aborted and the whole execution may start from the beginning.

  Transactions provide atomicity, consistency and isolation: all other threads will observe the results of the successful transaction and never an intermediate situation. Furthermore, transactions must be serializable, i.e. they appear to execute sequentially, in a one-at-a-time order. When a part of code needs to be executed atomically, it only have to be defined into a transactional block as shown in figure 2.14.

```
transaction_begin()
        ...
        ...
        critical section
        ...
        ...
transaction_end()
```

Figure 2.14: Critical section is being protected by a transaction block

With these constructs in place, transactional memory provides a high level programming abstraction by allowing programmers to enclose their methods within transactional blocks. The data within the transaction are protected from external conflicts, ensuring that either the operations will complete or no action is taken at all. Unfortunately, concurrency related bugs are still possible in programs that use a large number of transactions. These bugs can often be difficult to debug since breakpoints cannot be placed within a transaction.

TM comes in two main implementation formats, software transactional memory (STM) and hardware transactional memory (HTM). STM implements transactional memory exclusively in software. Many programming languages such as C/C++, Common Lisp, Java, Haskell, Python, Scala etc., provide libraries with STM implementations. STM is by nature more flexible than HTM, easier to change and evolve,

and has no need of any special hardware support. However, STM implementations suffer from high overhead as the procedure of detecting conflicts and bringing the system back in its previous state when an abort occurs is time-consuming and can degrade performance.

On the other hand, HTM implements transactions fully in hardware. HTM was developed to overcome constraints of software transactional memory, taking advantage of processor's caches and buffers to keep meta-data and logs in cases of abort, as well as cache coherence protocols to detect conflicts between transactions. HTM is limited by hardware resources, like for example the cache size. Furthermore, the instructions of HTM are not the same in processors that support different HTM implementations and as a result, a program should be rewritten to execute on another HTM platform.

Finally, there are also combined implementations such as: Hybrid transactional memory (HyTM) which supports HTM execution, but when hardware resources are exceeded, falls back on STM, and Hardware-assisted STM (HaSTM) which combines STM with new architectural support.

2.4.6 Basic Data Structures

A data structure is an efficient way to organize, save, manage and operate on a set of data. More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data. Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, relational databases commonly use *B-tree* indexes for data retrieval, while compiler implementations usually use *hash tables* to look up identifiers. Nowadays, data structures are used to efficiently handle enormous amounts of data in cases of e.g. large databases and internet services. As the performance of an algorithm is highly related to the underlying data structure, the creation of structures with the highest possible performance is a crucial issue.

As uniprocessors systems are being abandoned day by day, data structures should be redesigned to operate on multicore machines. In these machines, multiple threads are accessing shared data simultaneously resulting in concurrent data structures. These structures, when designed optimally, can exploit the capabilities of today's powerful multicore systems and result to fast and efficient algorithms. Unfortunately, concurrent data structures are far more difficult to design than sequential ones because threads executing concurrently may interleave their steps in many ways, each with a different and potentially unexpected outcome. This requires designers to understand new design methodologies, and to adopt a new collection of programming tools.

When designing concurrent data structures, programmers should be aware of the principles of parallel programming we have examined at subsection 1.2.5, that is correctness and progress and aim to high performance and scalability. On today's machines, the layout of processors and memory, the way data are organized in memory, the communication overhead between multiple processors, all influence performance. Due to the fact that

multiple threads access shared data and in order to ensure correctness, isolation of critical sections and consistency in the data structure, a synchronization technique is needed. As we have already mentioned, there are many synchronization models such as locking (either coarse-grained or fine-grained), atomic instructions, transactional memory. A selection of an inappropriate synchronization technique and a design that causes high contention on shared data between the threads, can not only lower the speedup of the concurrent algorithm as the number of cores increases but also degrade performance. As a result, programmers should have in mind the characteristics of each system and try to exploit parallelism as much as possible.

Nowadays, there is a huge amount of data structures, and even more concurrent implementations of them. We are now presenting *binary search trees*, *skip lists* and *priority queues*, on which we focused the most at this thesis.

2.4.6.1 Binary Search Tree (BST)

A binary search tree is a rooted tree, i.e. an undirected graph in which any two vertices are connected by exactly one path, and with one vertex designated to be the root. Each child node of the BST must either be a leaf node or the root of another binary search tree. The internal nodes store a key (and optionally an associated value) and have two distinguished sub-trees, commonly denoted left and right. The tree satisfies the binary search property, which states that the key in each node must be greater than or equal to any key stored in the left sub-tree, and smaller than or equal to any key stored in the right sub-tree.

Binary search trees are frequently used because they are easy to implement and their sorting and searching algorithms such as in-order traversal can be very efficient. They are used to construct more complex abstract data structures such as sets, multisets, associative arrays, heaps, hash trees, AVL trees, which are *balanced* binary trees, etc. Binary search trees keep their keys in sorted order, such that lookup and other operations can use the principle of binary search: when looking for a key in a tree (or a place to insert a new key), they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, to continue searching in the left or right subtree comparing the requested key with that of the examined node. On average, this means that each comparison allows the operations to skip about half of the tree, such that the complexity of each lookup, insertion or deletion is about *O(logn)*. At the scenario where at a BST with n nodes, each internal node has only one child-node, the tree degrades to an one-dimension array and operations take about *O(n)*.

There are many algorithms of concurrent binary search trees that utilize various synchronization mechanisms (from protecting the tree with a single lock to using atomic instructions and TM) in order to allow any number of concurrent threads to perform searching, insertion, deletion and rotation (reorganization or rebalancing) on the shared structure.

2.4.6.2 Skip List (SL)

Skip lists were evolved as an optimization to traditional linked lists, in order to reduce the time of searching a key within an ordered sequence of elements. As shown in figure 2.15, a skip list is a probabilistic data structure, built in layers. The bottom layer is an ordinary ordered linked list and each higher layer acts as an ”express lane” for the lists below, where an element in layer ***i*** appears in layer ***i + 1*** with some fixed probability ***p*** (usually p=0.5 or p=0.25). On average, each element appears in ***1/1-p*** lists, and the tallest element (usually a special head element at the front of the skip list) in all lists.

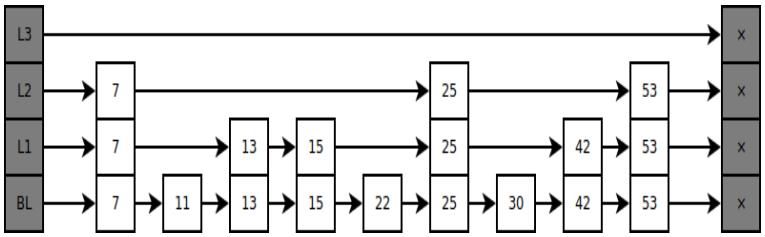


Figure 2.15: An example of a skip list with four layers

A skip list has three main operations: *search (element)*, *insert (element)*, *delete(element)*. As SL’s name implies, higher level lists allow skipping over many items and accelerate these operations. A **search** for a target element, which is the common basis of the two other aforementioned operations, begins at the head element in the top list, and proceeds horizontally until the current element is greater than or equal to the target. If the current element is equal to the target, the procedure succeeds and returns. In any other case, the search continues from the previous element at the slightly more complete list at one level lower. The expected number of steps in each linked list is at most ***1/p*** and therefore, the total expected cost of a search is ***O(logn)***, given that p has a constant value.

Delete operation has first to perform a lookup of the requested element with the procedure described above and then delete this element (if found) from every level of the SL it exists. Respectively, if an element is going to be **inserted**, lookup is used to find element’s future position and a new entry is inserted at the base level linked list. Then, the entry’s height is computed by using a ”flip coin” and as a result a node will be at the level l with probability 2^{1-l} .

Skip lists are the basis of many abstract data types such as priority queues and dictionaries, and they are used in Databases Systems, in statistical computations etc. Therefore, many concurrent SL implementations have been proposed [17, 20, 31, 27, 8, 12, 16], using either locks or atomic operations.

2.4.6.3 Priority Queue (PQ)

Priority Queues are abstract data types, i.e. they define the possible operations and values on data of this type without declaring their specific characteristics. PQ associates each element with a priority value, a score, and its entries are served based on their priorities: smaller scores mean higher priority. A priority queue typically provides a **findMin()**

method to find the item of minimal score (highest priority), an **insert()** method to add an item to the set and a **removeMin()** (or **removeMax()**) method to remove the minimal (or maximum) score element.

Priority queues can be naively implemented with ordered or unordered arrays and linked lists or more efficiently with heaps or skip lists:

- **PQ implemented with binary heap:** A heap is a tree based data structure in which all the nodes of tree are in a specific order. The binary heap has two common variations: the min heap, in which the smallest key is always at the front, and the max heap, in which the largest key value is always at the front. It can be used to implement a priority queue at which the search for the element with the minimum (or maximum) score will take O(1) time and the removal of this element will take O(logn) time as well as the insertion of a new element.

- **PQ implemented with skip list:** Skip lists have been described previously. The minimum priority element of the PQ will be the first element of the skiplist. The complexity of PQ's operations is equal with that of the skiplists.

Priority queues are used in graph operations such as in Dijkstra's Algorithm to extract the minimum efficiently, in Prim's algorithm to find the minimum spanning tree of a connected and undirected graph (implementation with min-heap provides better results in this case) in Best-first search algorithms, like the A* search algorithm in order to to keep track of unexplored routes; the one for which the estimate of the total path length is smallest is given highest priority. Furthermore, PQs can be used to manage limited resources such as bandwidth on a transmission line from a network router, for load balancing on servers, for interrupt handling and generally it can be efficient in cases where high-priority operations may occur and need to experience lower latency.

Due to their high applicability, there have been many concurrent implementations proposed for priority queues [2, 3, 6, 22, 24, 28, 30, 33, 35, 36] especially with skip lists as the underline data structure. One of the main constraints in designing parallel priority queues is the *deleteMin* operation, which leads many threads to read and write at the same memory locations, creating high contention between them, something that degrades performance and scalability. In order to overcome this constrain, **relaxed** concurrent priority queues have been proposed (e.g. at [2]) at which, threads instead of trying to delete the unique element with the minimum (or the maximum) score, they are deleting a random element from a set of entries with the lower (or higher) scores, resulting in less intensive algorithms.

2.5 Motivation of experimenting with priority queues

At this thesis we evaluate the performance and scalability of concurrent data structures on NUMA systems. The characteristics of non-uniform architectures do not affect all data structures with the same manner. In order to obtain a ruff idea about the behavior of concurrent data structures in NUMA, we experimented with Ascylib library [10]. This

library contains a collection of sequential, lock-based and lock-free implementations for data structures such as: linked lists, hash tables, skip lists, binary search trees, priority queues, and stacks.

More specifically, we experimented on a NUMA server with 4 sockets and 8 physical cores at each node. This server uses hyperthreading technology, according to which, OS regards each physical core as 2 cores, allowing two processes to be scheduled simultaneously at the same physical core. Therefore, the number of hardware threads is double the number of cores, resulting to 64 hardware threads on our NUMA server.

The data structures that we chose to work with are: Binary Search Trees, Skip Lists and Priority Queues. We now present the results of ASCYLIB implementations for these data structures in our NUMA server. The metric used for our evaluations is throughput, i.e. the number of operations executed by the whole set of threads per unit of time. Throughput is measured in Mops/sec (Millions of Operations per Second) and is calculated as the sum of each thread's operations and divided by the duration of execution. For the sake of consistency of our results we calculated the average of many executions of the same configuration and used thread placement to "pin" threads to cores. Table 2.1 describes the policy with which threads are pinned to the four numa-nodes (Ht stands for Hyperthreading).

Number of Threads	Numa Node
1-8	0
9-16	1
17-24	2
25-32	3
33-40	0 (Ht)
41-48	1 (Ht)
49-56	2 (Ht)
57-64	3 (Ht)

Table 2.1: Distribution of threads at the four sockets of the NUMA server. When Ht is presented, Hyperthreading mechanism is activated.

For the following diagrams, the initial size of the structure at each implementation is 1024 key-value pairs, and threads are operating at the structure for 5 seconds. During this time threads perform enough operations to get stable results. Each time a thread has to make an operation, generates a random key in set [0...2048] and according to the given configuration of update and lookup percentages, either searches for that key, tries to insert

or delete that key. The configuration of the diagrams below is 80% search, 10% insert, 10% delete operations.

- **Binary Search Trees**

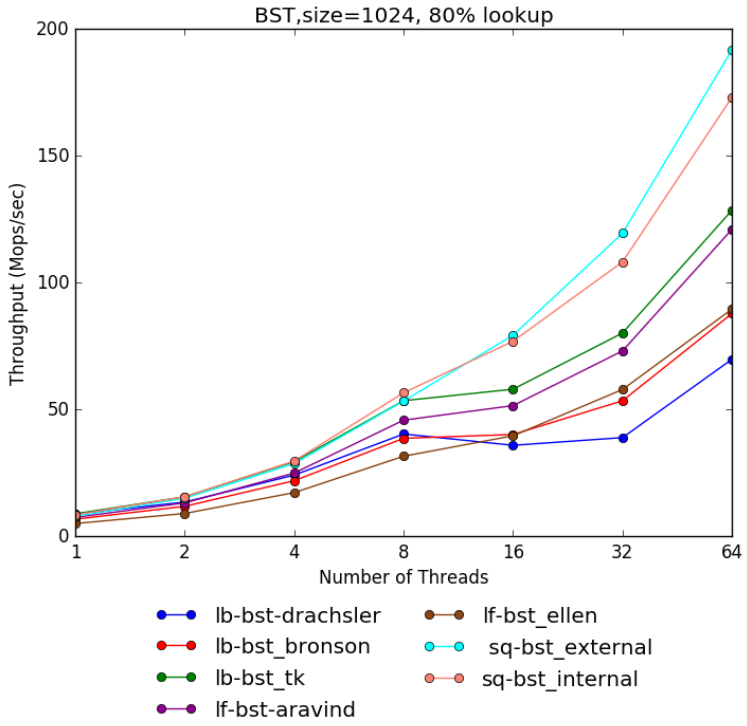


Figure 2.16: Ascylib's concurrent implementations of BST. The update ratio is 20% equally separated at insert and delete requests.

At the diagram 2.16, we evaluate the following concurrent binary search trees:

- **lb-bst-bronson:** Lock-based concurrent partially external binary search tree proposed by Bronson et.al. [4]
- **lb-bst-tk:** Lock-based concurrent binary search tree proposed by David et.al. [10].
- **lf-bst-drachsler:** Lock-free concurrent internal binary search tree proposed by Drachsler et.al. [13].
- **lf-bst-aravind:** Lock-free concurrent external binary search tree proposed by Nararajan et.al. [23].
- **lf-bst-ellen:** Lock-free concurrent external binary search tree proposed by Ellen et.al. [15].

Sq-bst-external and sq-bst-internal are sequential implementations of external and internal binary search trees respectively, do not use any synchronization method

and are an upper bound of performance. Both the lock-based and the lock-free implementations present high scalability, and they do not seem to be affected by the numa effect.

- **Skip Lists**

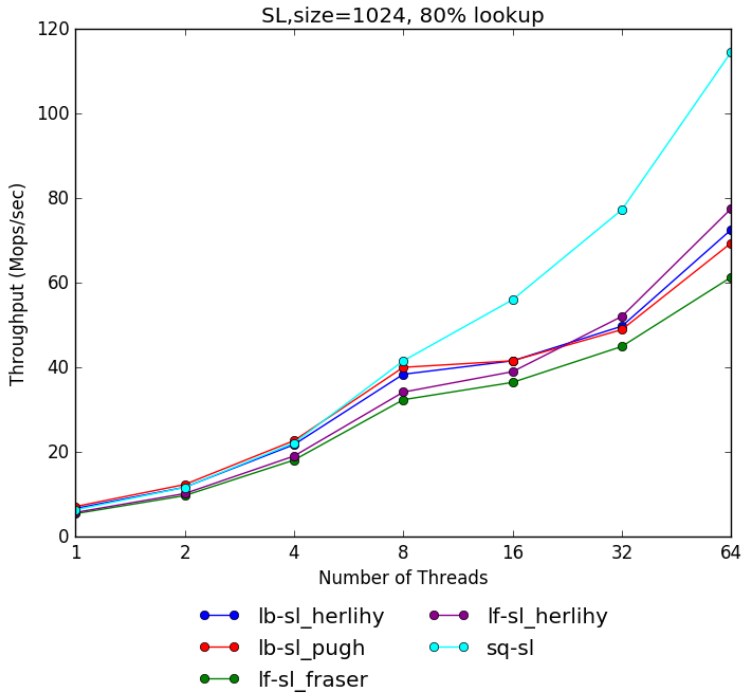


Figure 2.17: Throughput of Ascylib's skiplist implementations. The operation workload is: 80% lookup, 10% insert, 10% delete.

At the diagram 2.17 we evaluate the following concurrent skip lists:

  - **lb-sl_herlihy:** Lock-based concurrent skip list proposed by Herlihy et.al. [20]
  - **lb-sl_pugh:** Lock-based concurrent skip list proposed by Pugh et.al. [27]
  - **lf-sl_herlihy:** Lock-free concurrent skip list with wait-free contains operator proposed by Shavit et.al. [31]
  - **lf-sl_fraser:** Lock-free concurrent skip list proposed by Fraser et.al. [17] using Herlihy's optimization

Sq-sl is a sequential implementation of a skip list without any synchronization method and is used as an upper bound threshold of throughput. We observe that throughput increases when the number of core increases. However, when using more than one numa nodes, the scalability degrades.

- **Priority Queues**

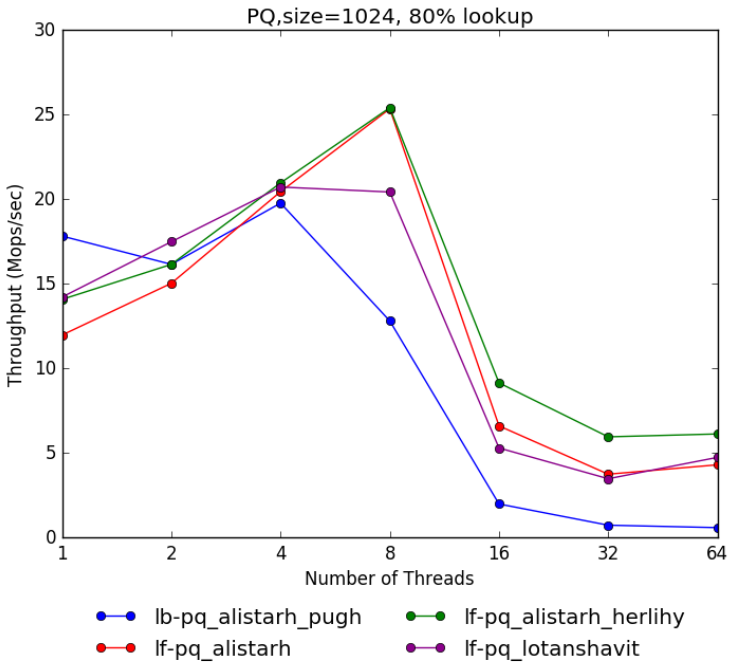


Figure 2.18: Ascylib's concurrent algorithms for PQ, where the percentages of insert and deleteMin are 10% each. High contention between threads leads to performance degradation.

At figure 2.18 we evaluate the following concurrent priority queues:

- **lf-pq-lotanshavit** Lock free concurrent priority queue proposed by Shavit et.al. [31].
- **lb-pq-alistarh-pugh:** Lock-based concurrent priority queue proposed by Alistarh et.al. [2] based on Pugh's skip list.
- **lf-pq-alistarh:** Lock-free concurrent priority queue proposed by Alistarh et.al. [2] based on Fraser's skip list.
- **lf-pq-alistarh-herlihy:** Lock-free concurrent priority queue proposed by Alistarh et.al. [2] based on Herlihy's skip list

While the threads operating at the priority queue remain at the same socket, most of the implementations (except from the lock-based) scale quite well, but when the number of threads exceeds the number of physical cores, we observe a huge degradation at throughput. This is caused mainly by the fact that threads content for the same memory locations due to the deleteMin operation. This leads to the appearance of numa effect, i.e. the "ping pong" of the relative cache lines between the sockets.

Figure 2.19 visualizes the numa effect. The algorithms examined are lf-pq-alistarh-herlihy, and a sequential implementation that does not use any synchronization mechanism. At the lock-free implementation, all threads are pinned to cores of the first numa-node using hyperthreading mechanism when needed. The implementation sequential_2_sock uses at the beginning the 8 physical cores of socket 0 and then the 8 physical cores of socket 1. Finally, sequential_4_sock uses the first 2 physical cores of socket 0, then the first 2 physical cores of socket 1, etc until socket 3 and then again socket 0,1,2 and 3. While the lock-free implementation presents a scalable behavior, the sequential ones are affected by the memory traffic between the different sockets, making obvious that the numa-effect can cause significant performance degradation even when the contention between threads is low.

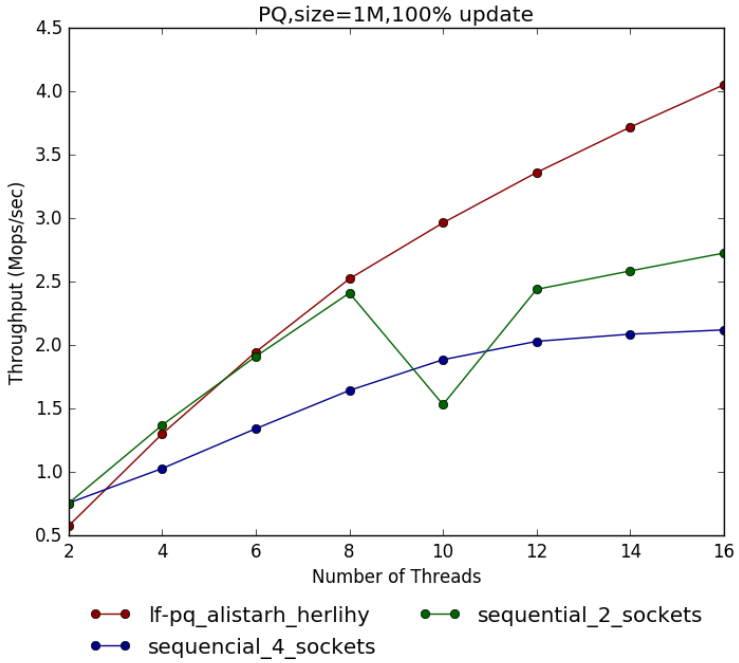


Figure 2.19: Visualization of numa effect: higher throughput is achieved when a single socket is used.

Consequently, since the performance of priority queues is remarkably affected by the numa effect, this thesis is focused on this type of data structure. We evaluated techniques that can be used to overcome constraints non-uniform systems impose, and adapted state-of-the-art PQ algorithms in order to create a numa-aware priority queue.

Chapter 3

Numa Aware Data Structures

3.1 Introduction

As we have discussed, despite the fact that NUMA allows a system to scale to hundreds of cores, interconnection network's high latency and limited bandwidth can drastically degrade performance. Methods described at 2.3.2 aim to limit the effects of remote memory accesses and cache invalidations and can be really helpful in many cases. However, they do not guarantee the elimination of numa effect and sometimes their implementation is challenging. To maximize performance and exploit the parallelism of multicore NUMA systems, programmers turn to the design of numa-friendly algorithms. Due to the fact that concurrent data structures are widely used by these algorithms, researchers focus on the design of numa aware data structures. At this chapter, we delve into the main characteristics of numa aware data structures and we present some state-of-the-art related algorithms.

The main challenge when designing any concurrent data structure is how to deal with thread contention on shared data. In a shared memory system, when multiple threads consequently send write and read requests, the traffic of the memory bus is increased due to the cache coherence protocol. Therefore, at a NUMA machine, when the contention between the numa nodes is high, i.e. threads from different sockets send read and write requests for the same cache lines, copies of them are being created at different numa-nodes and the interconnection network between the sockets is contented due to MESI protocol. This results to performance degradation and scalability limitation. Consequently, researchers focus on cases with a large amount of update operations and attempt at least to maintain a constant performance as the number of threads exceeds the number of cores on one socket.

To design a scalable data structure for non-uniform architectures, one should have in mind the characteristics of numa systems that can degrade performance. More specifically, it is important for the data structures to reduce communication across NUMA nodes, and to reduce accesses to remote memory. The main characteristic of NUMA is that it provides very low latency operations when accessing node-local data but higher-latency at remote memory accesses. Therefore, memory accesses should be limited inside the socket-local memory.

Furthermore, the performance of an application is affected by the way threads are assigned to the cores of the system. Under certain circumstances, some implementations will allow a task to be executed on another processor. For example, when two processor-intensive tasks (A and B) have affinity to one processor while another processor remains unused, many schedulers will shift task B to the second processor in order to maximize efficiency. Task B will then acquire affinity to the second processor, while task A will continue to have affinity to the first core. Despite the fact that this technique helps in reducing load unbalancing, it may cause a rapid increase at the number of cache misses. Especially on non-uniform architectures where cache misses are costly, a thread is usually pinned to one core and it is not migrating between the sockets.

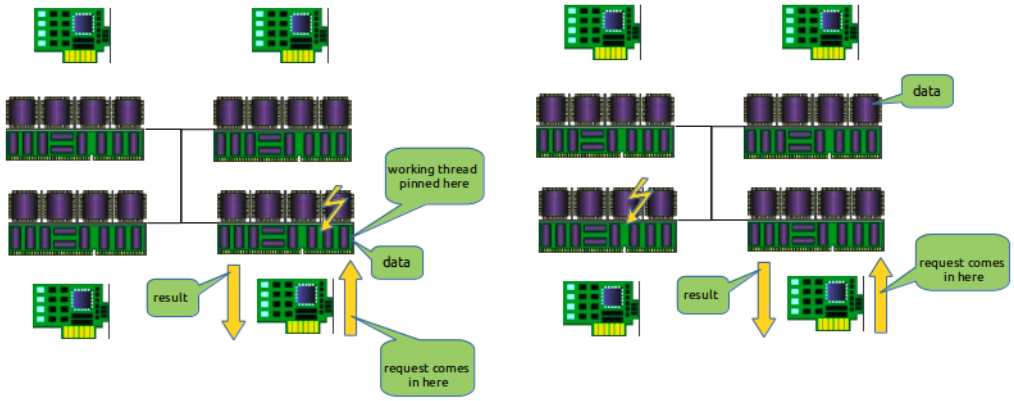


Figure 3.1: Numa-aware vs non-numa-aware execution. At the first case the working thread is pinned to a specific core and accesses node-local data while at the second case the working thread is migrating between the cores and accesses remote data.

Figures 3.1a and 3.1b visualize the main differences between numa-aware and non numa-aware implementations as described above: working threads are pinned to specific cores and access *node local* memory. The algorithms presented at the following sections are based on these attributes and lead to numa-friendly approaches. We are going to delve into these approaches, starting with numa-aware locks that can be used to create blocking data structures. We discuss the ideas of combining and delegation, we present ways of adopting them to create concurrent data structures, and we study state-of-the-art numa aware algorithms for concurrent skip lists and priority queues.

3.2 Numa Aware Implementations

3.2.1 Lock Cohorting

Before analyzing various algorithms for numa aware data structures, it is worth to mention the technique of lock cohorting. Dice et al. [11] proposed a way to implement numa-aware locks and transform any spin-lock algorithm, into a scalable NUMA-aware lock-based implementation. More specifically, every spin-lock or spin-then-lock can be transformed into a numa-aware lock with a little overhead, allowing sequences of threads into the same socket, to execute consequently.

This algorithm exploits the idea of lock cohorting, i.e. when a thread releases a lock, it can detect if there is a non-empty cohort of threads concurrently attempting to acquire the lock (*cohort detection property*). In more detail, researchers propose the existence of a *thread-oblivious* global lock G and associate each numa-node i with a distinct local lock Si that has the cohort detection property. Locks S and G can be of different types: figure 3.2 demonstrates a case of the lock-cohorting algorithm where G is a simple test-and-test-and-set backoff lock (BO) and S is a MCS queue lock.

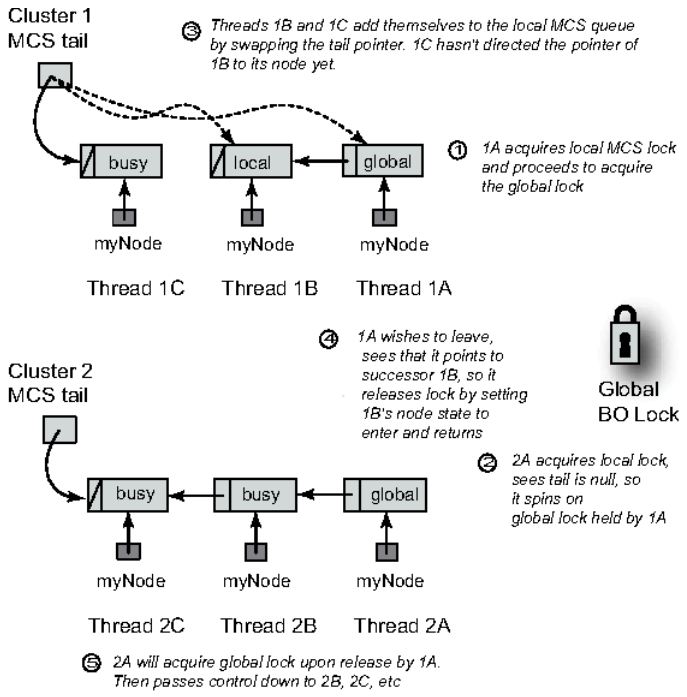


Figure 3.2: Lock Cohorting example for a system with two numa nodes

To access the critical section, a thread must hold both the local lock Si of its socket, and the global lock G. When a thread wishes to enter the critical section, it first acquires its local lock, and according to its state decides if it can immediately enter the critical section

or must compete for G. When a thread leaves from the critical section, it checks if there are any threads of the same socket waiting on Si. If so, the exiting thread will release Si, without releasing G, setting the state of Si to *local release*. The thread that manages to successfully re-acquire Si will enter the critical section. On the other hand, if there are no threads of the same numa-node waiting for the locks, the exiting thread will release both Si and G, setting the state of Si to *global release*. This indicates to the next local thread that it must re-acquire G before it can enter the critical section.

When a thread of socket 0 has acquired the global lock, its control is subsequently passed among contending threads within the socket 0, resulting to low-cost transitions of threads into the critical section, and minimizing lock migration between the sockets. In order to prevent threads of other sockets from starvation, while the global lock is acquired by threads of socket 0, there should be obtained a relative policy, e.g. to release the global lock after an allowed number of consecutive accesses. The algorithm described above can fit all combinations of thread-oblivious and cohort-detection-capable locks, with some of them outperforming others. In general, cohort locks perform as well or better than known locks when thread-contention is low and significantly exceed them as the contention increases.

3.2.2 Combining and Delegation

3.2.2.1 Delegation

Delegation, in computer science, refers to one entity passing something to another entity. At this subsection we present the idea of delegation and two algorithms that exploit this idea to create numa-aware data structures. In delegation implementations, only one of the threads can access the shared data: this thread is called *server*. All other threads, which are called *clients*, are not allowed to make operations on the shared data: when a client thread wishes to access the shared data, it makes a request to a server and waits for a response. At the most cases, server is pinned to a specific core, receives requests from the clients, executes these requests to the data on clients' behalf and sends the results back to them. Therefore, there is no need for synchronization since only one thread , the server thread, accesses the data structure.

Delegation is more appealing to be implemented using message passing between threads. However, message passing is not provided by hardware at many architectures, generating the need for an implementation over cache coherent shared memory. Server and clients are communicating by reading and writing at shared cache lines, activating the cache coherence protocol which can become a bottleneck in a poorly designed algorithm. Petrović et. al. [26] study the effect of cache coherence mechanisms at delegation and propose methods to optimize this technique. Delegation can assist at creating numa-aware data stuctures, as long as its implementation is done with regard to the characteristics of non-uniform architectures. It is an efficient and relatively simple synchronization technique especially for data structures that are not easily partitioned such as priority queues, stacks, etc. As we will see later, delegation at these data structures can outperform state-of-the-art numa-oblivious fine-grained or lock-free algorithms under high thread contention.

3.2.2.2 Combining

We present at this subsection the idea of combining, as it was analyzed at [19]. *Flat Combining (FC)* as mentioned at [19], is a form of delegation and is a basic component of many numa-aware data structures, especially of those that suffer from low parallelism such as stacks, priority queues etc.

The basic idea of flat combining is illustrated at figure 3.3. The main idea is that instead of using fine-grained locks to guarantee synchronization between threads, it is better to have a single thread holding a lock to perform the combined access requests of all others at the data.

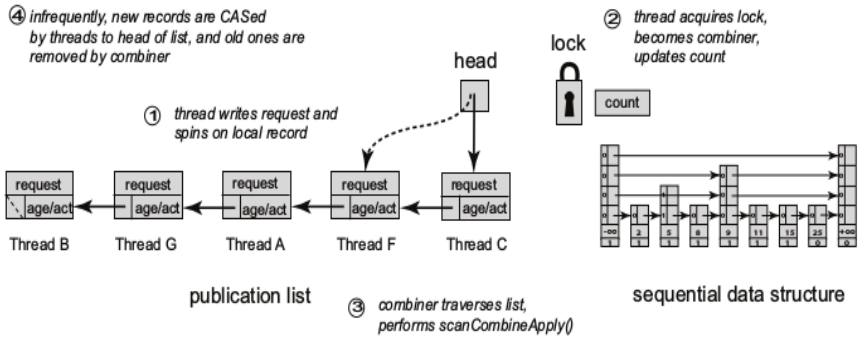


Figure 3.3: General view of a flat combining structure

Flat Combining algorithms have a sequential data structure protected by a single global lock, and an associated dynamic publication list with size k*N where N is the number of threads. Each thread that wishes to access the data structure, has to add a unique entry at the publication list. Each thread will post its requests at the corresponding entry. After writing the request, a thread checks if the shared lock is free and if yes it tries to acquire it, using an atomic operation like CompareAndSwap. Only one thread will finally manage to acquire the lock and become the *combiner*.

The combiner scans the publication list, collects pending requests, applies combined requests to the data structure, writes the results back to each associated publication record and releases the lock. Threads that do not manage to become the combiner, spin at their records waiting for the response. The FC implementation guarantees that when a response is given, the corresponding operation has been executed at the data structure. Furthermore, a form of *linearizability* is created, as a thread will not post another request until a response for the previous request arrives. Therefore, even though some methods at the publication list may be missed by the combiner, a later thread cannot get a response unless the earlier missed requests are no more pending. In order to keep the size of the publication list relatively small, a cleanup operation is done at regular intervals and removes records that are no more used.

Flat combining implies that a single thread will access the data structure, reducing overhead due to the cache coherence protocol and the synchronization between threads.

This can be exploited at non-uniform architectures in order to reduce the amount of remote memory accesses. However, the design of a FC algorithm should be done carefully, having in mind that a stall at a combiner (due to a programming bug, a received signal etc.) can cause the delay of the whole set of threads. If a policy for this specific situation is obtained, FC guarantees to be *starvation-free*, as all requests will finally be combined and executed at the structure.

Flat combining is more effective at data structures where parallelism is relatively low, stacks, linked lists, priority queues etc., and more specifically where fine-grained synchronization techniques perform poorly. Given a data structure, and k operations on it, each taking time t, then flat combining can be applied if the time that it takes to combine and execute these operations are less than $k*t$. Roghanchi et al. [19] propose FC implementations of queues, stacks, priority queues based both on skip lists and on pairing heap that outperform known implementation. On the other hand, at structures where fine-grained locking can fully exploit parallelism, such as heaps, binary search trees, etc, combining is not a good candidate for implementation, as the performance is limited by the single combiner. In order to overcome this constraint, advanced forms of FC are necessary, allowing multiple instances of flat combining to operate concurrently.

3.2.2.3 NUMA-friendly stack with Elimination and Delegation

Calciu et al. [5] apply the idea of delegation to create a numa-aware stack. Stacks are the basis of many abstract data structures and algorithms. Therefore, their performance is a crucial subject for research. Unfortunately, stacks cannot be easily partitioned without violating their last-in-first-out (LIFO) property. Multiple threads attempt to update the top of the stack, resulting to high contention and consequent invalidations of cache lines. As we have seen, at non-uniform architectures this leads to a ping-pong of cache lines between the sockets, degrading performance.

The algorithm presented here is based on the ideas of delegation and elimination. There is one dedicated thread, *server thread* that accesses the stack. None of our known synchronization mechanisms is used, as there is only one thread operating on the data. Server thread receives push and pop requests from the clients, as shown in figure 3.4.

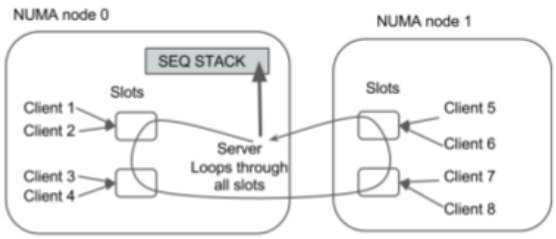


Figure 3.4: NUMA-aware stack using elimination and delegation at two numa-nodes

There is a specific number of slots available that are distributed to threads of a socket. The server loops through all the slots, collecting and processing requests and writing the

responses back to the slots. In order to enable more parallelism, elimination is used as follows: when a client thread wishes to make an operation, it first tries to eliminate, finding a reverse method (push-pop) using e.g. the *randezvous* method described at [1]. If elimination fails, client delegates its operation to the server, i.e. posts its operation to a slot and either spin-waits for the result (pop method) or not (push method). Elimination can be implemented using a special data structure (e.g. an array) per NUMA-node in order to avoid the overhead at the interconnection network. If the percentages of inverse operations are roughly equal, many pairs of threads can exchange arguments eliminating their requests and as a result the load at the data structure is decreased.

Since accesses at the stack are serialized, the throughput achieved is limited to the single-threaded execution (server thread) and this can be a limitation for the application's performance. Furthermore, when threads are not so active, i.e. the number of requests that access the stack is relatively small, the overhead of the server-clients communication is higher and can degrade scalability. However, elimination and delegation help to reduce traffic at the interconnection network as well as thread contention at the data, outperforming many concurrent stack algorithms under high contention. This combination of techniques can be really effective at structures like lists, queues, etc. where there is a significant amount of operations that can be eliminated.

3.2.3 The adaptive priority queue with Elimination and Combining

The following algorithm was not designed especially for non-uniform architectures but can be utilized for this goal. Calciu et al. [6] propose a linearizable skip list based Priority Queue that utilizes Elimination and Combining techniques. As shown at figure 3.5, the underlying skip list is splitted in two parts: *sequential* and *parallel*. The size of these parts can change (i.e. *adapt*) regarding to the contention between the threads and the kind of operations executed at the queue.

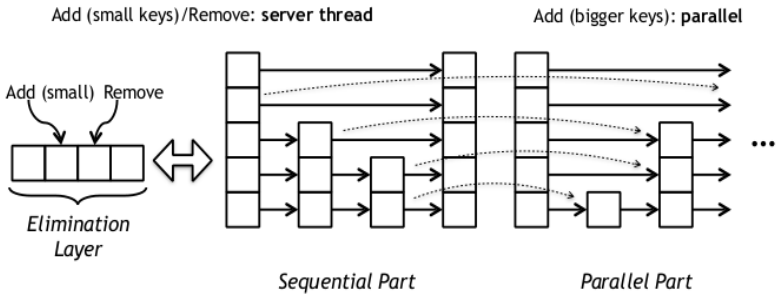


Figure 3.5: The underlying skip list is partitioned into a sequential (where elimination and combining are used) and a parallel part

The sequential part of the skiplist is used to serve *deleteMin()* operations at the priority queue, as well as *insert(k)* when *k* is relatively small (close to the queue's minimum). The

policy at this part is based on combining and delegation: operations are executed at batches by a dedicated server thread. On the other hand, *insert(k)* operations - when k is large - are going to be executed at the parallel part.

When a thread wishes to insert a key into the priority queue, it first checks if this key is greater than the maximum of the sequential part of the skiplist. If yes, then the key is considered relatively large, and the thread operates at the parallel part, where synchronization is maintained using a Single-Writer Multiple-Readers lock. Otherwise, if the key is small, the thread will try at first to perform elimination with a deleteMin() operation, using an elimination array, as shown in figure 3.5. When the key is smaller than the priority queue's minimum, the add() operation can *immediately* eliminate with an available deleteMin(), else it remains at the array waiting for an *upcoming* elimination. DeleteMin() operations can be matched if insert(k) are available at the elimination array. Elimination allows matching operations to complete without accessing the shared data structure, increasing parallelism and scalability.

Partitioning the skiplist into these two parts is done in order to fully exploit the possible parallelism of insert(k) operations when the elements that are going to be inserted are "away" from each other and can be added in parallel. At the same time, insert(k) and deleteMin() operations that did not manage to eliminate are executed sequentially by a server, reducing additional overheads. The borders between the two parts are changing according to the intensity and the type of requests at the priority queue. This technique can be utilized in order to create numa-aware data structures, as accesses to the sequential part are done by one server thread, minimizing cache invalidations and reducing interconnection traffic. The parallel part is accessed by many numa-nodes and can be a bottleneck at the performance. A possible but risky solution would be to split the parallel part in the available numa nodes.

3.2.4 Node Replication (NR)

The idea of memory replication can be particularly useful when designing numa-aware data structures, as it can help minimizing remote memory accesses. Calciu et al. [7] propose **Node Replication (NR)**: a *black-box* approach to convert sequential data structures to NUMA-aware concurrent structures satisfying linearizability.

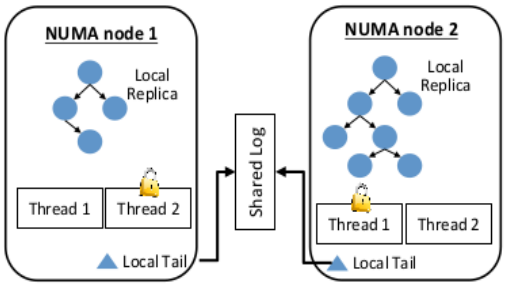


Figure 3.6: Node Replication algorithm at a system with two numa nodes

NR replicates the data structure at each numa-node, creating a *local replica* and uses a log, shared among the different sockets to coordinate the whole set of threads. More specifically, within the numa-node, NR uses *flat combining* in order to create batches of outstanding operations that affect the data structure. When a thread wishes to update the structure, at first posts its request and tries to become the *combiner*, as shown at figure 3.6. After contenting with the other threads, a thread manages to become the combiner at this particular node for an amount of time. The combiner is responsible for posting the aforementioned batches of requests to the shared log. Furthermore, the combiner thread has to bring the local replica up-to-date by examining the log, and execute the node-local requests to its local replica.

The shared log is a circular buffer (with limited size) that stores update operations on the data structure. Read requests are not published at the log as they do not affect the replicas of the other sockets. Read requests are executed directly at the local replica using a reader-writer lock for each node. The combiner acquires the lock in write mode when it wishes to modify the local replica, while reader threads acquire the lock in read mode. Therefore, reads and search operations can be executed more efficiently, and without overloading the shared log.

NR offers the ability to transform any sequential data structure to a numa-friendly concurrent one, without requiring any inner knowledge of the structure, and therefore is characterized as a *black-box* technique. The replication of data adds a space overhead: at a system with N numa-nodes, NR consumes N times more memory, and thus it is more suitable for smaller data structures. Except from the accesses to the shared log, the majority of memory accesses are node-local, and as a result NR can outperform non numa-aware implementations, especially at cases when thread contention is high.

3.2.5 NumaSK

Another algorithm that is based on the idea of memory replication is proposed in [9], and refers to a numa-aware implementation of a **skiplist**. As we have described at subsection 2.4.6, a skiplist is built in layers: the actual data are at the lowest level, while the other levels operate as an index layer that helps accelerating each operation.

NumaSK exploits memory replication, not at the whole structure but only at the index layer which holds the *metadata*, i.e. information that can be used to skip some nodes while traversing the skiplist. An independent replica of the index layer is created at each numa-node, and each thread traverses the index layer associated with the socket it resides. As a result, operations do not need to make remote memory accesses and traverse the interconnection network between the sockets during e.g. a lookup of an element.

Figure 3.7 visualizes NUMAsk implementation at a four-socket system. As we can see, there are actually three layers: two are independent and local to each node, while the third is shared among the four sockets. The last layer is our known data layer, while the top layer is the index. NUMAsk defines an *intermediate layer*, which is a NUMA-local view of the actual shared data, it servers as a local base for update operations and helps in fully exploiting NUMA-local memory.

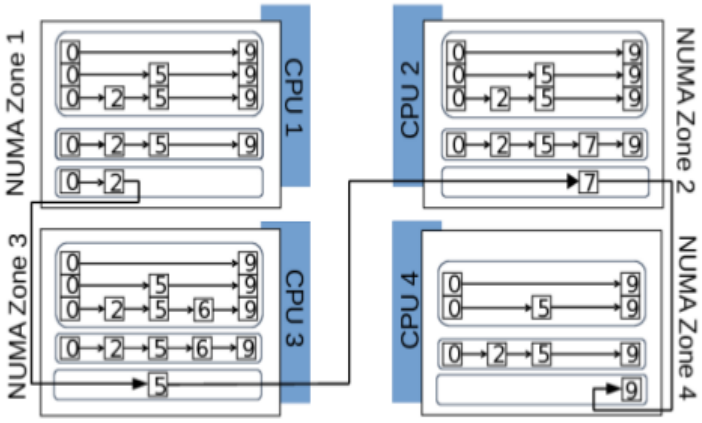


Figure 3.7: NUMASK algorithm on a server with four sockets. The data layer is at the base level and it is shared to all sockets. Index and intermediate layers are local to each numa-node.

When a thread wishes to make an operation, first traverses the local index layer replica until it finds a pointer to a node at the local intermediate layer. Then, the intermediate node will redirect the thread to reach the desired node at the data layer, where the actual operations are performed. We can see from figure 3.7 that node-local index and intermediate layers are not always consistent with shared data layer. This relaxed policy does not seem to affect the structure's correctness, as every change to the actual shared data will eventually be propagated to the node-local replicas using specially assigned per-node helper threads. Similarly to the majority of numa-aware implementations, NUMAsk outperforms other algorithms when thread contention is high.

3.3 Fast Fly-Weight Delegation (FFWD)

At this section we present *ffwd*, an efficient design of delegation that aims to optimize the communication between the server and clients, offering low latency and high throughput. Ffwd, as presented in [29] is one of the fastest delegation systems so far and is the basis of our implementation proposed at the next section.

To overcome the single-server throughput, multiple request/response pairs need to be exchanged in parallel. The performance of a delegation algorithm can be degraded due to many factors. The bandwidth and latency of the interconnection network, i.e. the amount of cache lines that can be transferred per second and the total time taken from the posting of a request to the arrival of the response by the corresponded client, can effect the delegation system. Furthermore, the overhead of cache coherence or messaging protocol, the speed at which requests are processed etc. define the behavior of a delegation algorithm. In order to counterbalance the limitation of one thread accessing and operating on data, the aforementioned factors should be kept in mind when working with delegation.

The layout of ffwd is illustrated at figure 3.8. The communication between the single

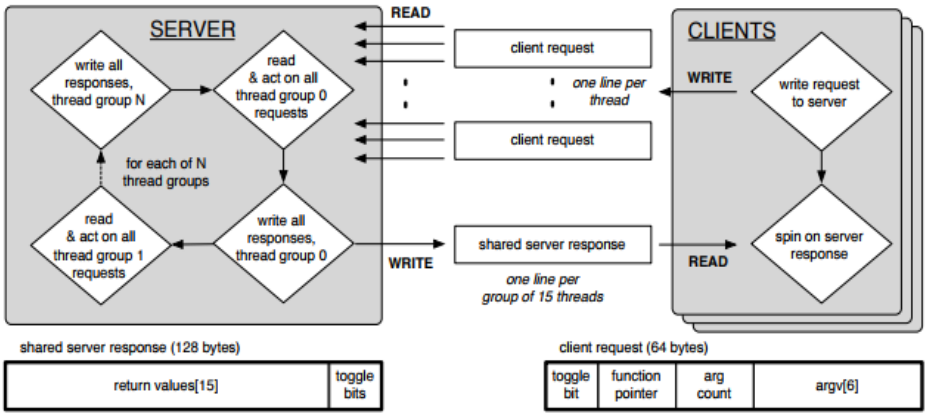


Figure 3.8: Fast Fly-Weight Delegation: Communication between server and clients is achieved using a specific amount of cache lines to reduce cache coherence traffic.

server and the clients is achieved reading and writing on shared cache lines. Each client core maintains a cache line pair of a specific size, where it posts its request. Server reads clients' requests, and writes back the responses. In order to reduce the communication overhead and the number of cache invalidations, the clients at each numa-node are reading the server's responses from the same cache line pair. Server thread is responding to a specific number of clients per socket and therefore, the response cache lines consist of a single slot per client. Each clients spins at its slot until the server writes the proper response. Consequently, request cache lines are written exclusively by clients and response cache lines are written exclusively by the server.

The server scans the first socket, reads and executes the upcoming requests (no locks are required as the server is the only one accessing the data), buffers individual responses locally until processing for the current socket is finished, writes the whole set of responses to this node's cache line pair and then proceeds to the next group of clients. As shown at figure 3.8, each request consists of a toggle bit, the function that has to be executed, an argument counter and the appropriate arguments. The process of a request is done as follows: the server loads the arguments provided into the appropriate registers, and calls the specified function. The response is a set of return values, each of them corresponding to a specific client, while toogle bits illustrate the state of requests (pending or completed).

Ffwd manages to optimize the idea of delegation, reducing the cache coherence communication at the interconnection network. When the contention and cache invalidations are high, ffwd can outperform lock-based and lock-free implementations. Threads are pinned to the cores and all allocations are numa-aware (done at the node-local memory), thus reducing remote memory accesses. A cache line pair is allocated for each client core, thus there is no contention between the threads for the posting of requests: the only contention is caused due to the server who is reading these cache lines. Furthermore, responses are buffered locally at server and then they are packed and written at a pair of cache lines that are shared between the cores of a socket. The first read by a client will bring the

response lines at socket's cache, from where all other reads will be served. However, the performance of ffwd is limited by the single server who is accessing the data structure, and therefore it is a better design choice for data structures that perform better on a single thread than on multiple threads. When the concurrency level is high, a fine grained method can lead to better results.

3.4 Numa Node Delegation (NUDDLE)

Ffwd offers an interesting approach to delegation using and monitoring a specific amount of cache lines, and as a result minimizing the cost of communication between the clients and the server and reducing the redundant traffic at the interconnection network between the sockets. However, the performance of any ffwd algorithm is limited by the rate with which a single thread can serve the upcoming requests and executes the corresponding operations. Ffwd outperforms non-numa-aware algorithms when the contention between threads is high, as the shared data remain at the server's node local memory and numa-effect can be minimized. Therefore, it would be useful if we could keep the shared data at a single socket's local memory, while trying to exploit the maximum possible parallelism.

At this section we propose an extension of ffwd algorithm called *numa node delegation*, that can be used to turn any arbitarry concurrent data structure into its corresponding numa-aware version, exploiting a satisfying amount of concurrency. The basic idea of NUDDLE is illustrated at figure 3.9 and it is that multiple servers inside the same numa-node can be used to process all upcoming requests.

In order to reduce numa-effect, it is preferable that threads that access the shared data structure are located at the same numa-node, and the structure is allocated at the socket-local memory. Therefore, the maximum number of servers that NUDDLE uses is equal either to the number of physical cores of a single numa node, or twice this number, if hyperthreading is enabled and more parallelism is desired. These servers are processing clients' requests according to the manner ffwd implies and operate at the data structure, which needs a synchronization method as it is accessed concurrently.

More specifically, NUDDLE algorithm operates as follows: all threads are pinned at specific cores and they are not migrating during the execution. One of the numa-nodes contains the servers, while the rest physical cores are available for clients. Each client, depending on the numa node that it belongs, sends requests to a specific server, using the message passing interface described in [29]. The client spins at its local slot and will sent again a request after receiving the response for the previous one. Each server is responsible for specific numa nodes: servers first check for upcoming requests, execute the proper operations, and write back the results.

In order to allow multiple servers access the same data structure, an underlying concurrent algorithm is needed, to provide correctness and progress at the whole execution. Every concurrent algorithm can be used as the underlying data structure, either lock-based of lock-free. At this thesis, we focused on the implementation of a *concurrent priority queue*, because it suffers from low scalability at non-uniform memory systems as we have

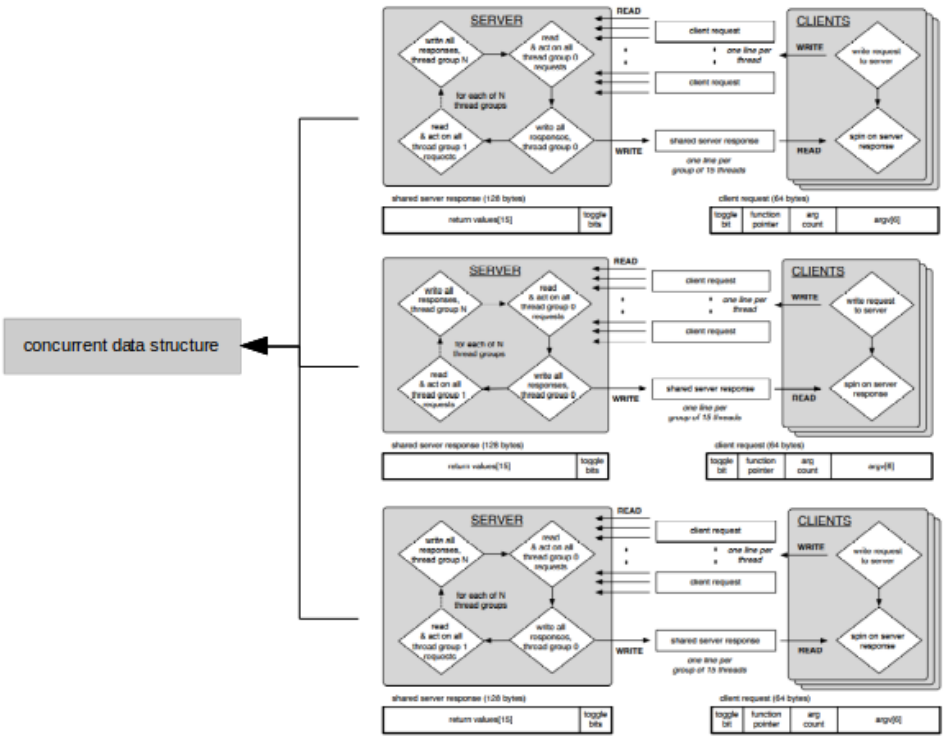


Figure 3.9: NUDDLE: multiple servers exist and process the clients' request, operating on a **concurrent** data structure

seen at section 2.5. We used the concurrent relaxed priority queue as described at [2], designed by Alistarh, based on Herlihy's non-blocking skiplist, as the underlying parallel algorithm. This is a lock-free implementation, based on atomic operations to offer synchronization, and can be used to implement the main abstract operations at a priority queue: insert(key), lookup(key), deleteMin(). The relaxed nature of this algorithm relies on the fact that deleteMin() does not return the single minimum element, but an an element among the first $O(p * (logp)^3)$ in the list, with high probability, where p is the number of threads. This approach reduces the contention between the threads and provides higher levels of parallelism. We used this algorithm among the other PQ implementations from ASCYLIB as it reached the highest performance, among the others. However, any concurrent data structure implementation that does not scale well at NUMA can be used as the underlying data structure, while other implementations such as binary search trees, hash maps etc. will not be benefited by NUDDLE as the parallelism will be reduced to the cores of a single socket.

NUDDLE is actually a **black box** technique: it can be used to create a numa-aware version of any concurrent algorithm. Correctness and progress of NUDDLE are guaranteed both from the concurrent data structure and from the way ffwd functions. Our implementation combines the high level of parallelism that a concurrent algorithm can

provide with the delegation technique, optimized to reduce the cost of communication between clients and servers. Threads are pinned to specific cores and all memory allocations are node-local, as described at chapter 2, in order to reduce remote memory accesses. However, the concurrency of NUDDLE is limited by the number of servers, as they are the only that access the data structure. As the communication overheads are minimized, the performance of a NUDDLE implementation is theoretically expected to reach a peak when all servers access the data structure and then is expected to remain constant. When numa-effect is intense, and the performance of concurrent algorithms degrades, NUDDLE offers an efficient alternative, which outperforms all other implementations. When parallelism is high and the cache invalidations not so often, limitations of NUDDLE may lead to lower throughput than the other implementations. Therefore, as we will see later, NUDDLE should be used when the contention at the shared data structure is high.

Chapter 4

Evaluation

At this chapter we present the evaluation of priority queues implementations at two NUMA platforms. We evaluated the ASCYLIB's concurrent priority queues, FFWD and NUDDLE implementations, with various configurations and we state here the most notable results. At the beginning, we analyze the experimental setup of our evaluation, then we present our results and finally we present the reasons that intensify the need for smart data structures.

4.1 System Configuration

At this section we present the architecture of the NUMA platforms we experimented on. We name the servers that we used for our experiments as *sandman* and *numascale* respectively.

The figure 4.1 is a simple illustration of the first NUMA platform, the architecture of which is presented at table 4.1.

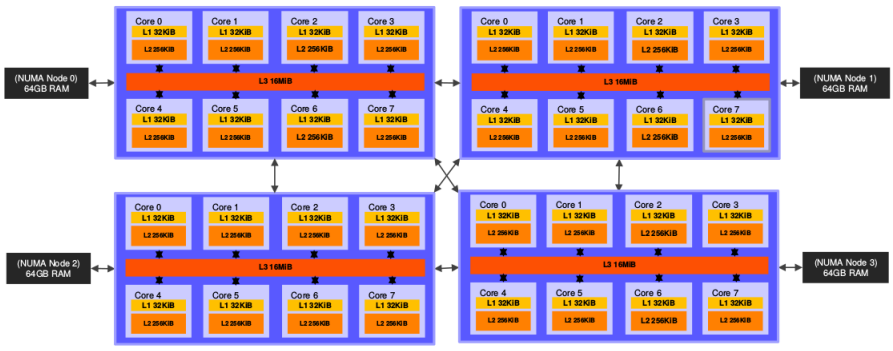


Figure 4.1: Sandman Architecture Layout

Reference name	sandman
CPU	Intel Xeon E5-4620
Processor Base Frequency	2.20 GHz
Number of Cores	4 * 8
Number of Threads	4 * 16
L1(data) cache/core	32 KB, 8-way, 64b block size
L2 cache/core	256 KB, 8-way, 64b block size
L3 cache/socket	16 MB, 16-way, 64b block size
RAM	256 GB
OS	Debian 8.8
Linux kernel	4.0.4

Table 4.1: Characteristics of the 1st NUMA platform.

In order to generalize results, another NUMA machine was used. As figure 4.2 shows, this machine consists of 6 physical servers: each server contains 3 AMD processors and each processor is separated into 2 numa nodes.

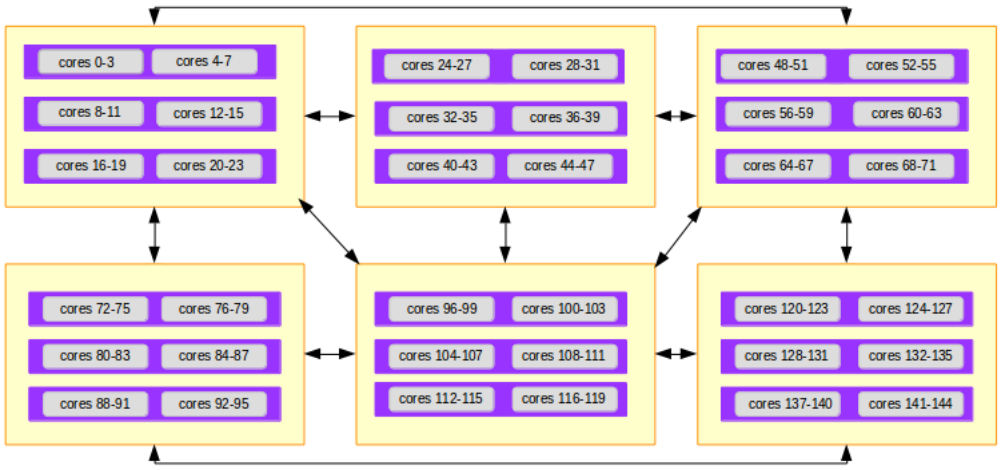


Figure 4.2: Architecture of the 2nd NUMA platform

Figure 4.3 illustrates the internal architecture of the AMD processor used. This platform's architecture is presented at table 4.2.

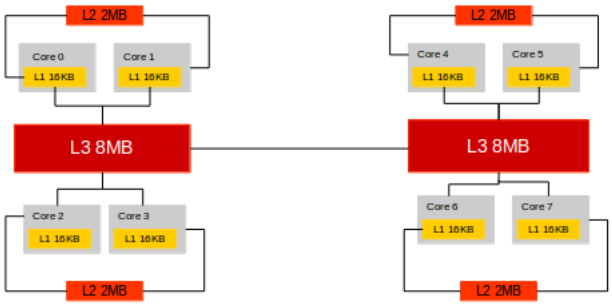


Figure 4.3: Architecture of an AMD Opteron 6328 processor

Reference name	**numascale**
CPU	AMD Opteron Processor 6328
Processor Base Frequency	3.2 GHz
Number of Cores	6 * 3 * 8
Number of Threads	6 * 3 * 8
L1(data) cache/core	16 KB, 4-way, 64b block size
L2 cache/two cores	2 MB, 16-way, 64b block size
L3 cache/numa node	8 MB, 48-way, 64b block size
RAM	2 TB
OS	CentOS Linux 7.5
Linux kernel	4.9

Table 4.2: Characteristics of the 2nd NUMA platform.

4.2 Results

4.2.1 Run Configurations

To evaluate the priority queue implementations, we experimented varying the size of the structure, the range of keys queried/used in operations, the kind of operations that access the queue, etc. More specifically:

- Threads are **pinned** to specific cores in order to receive more consistent results and to take advantage of locality provided by numa nodes, e.g, by sharing the same L3

cache. Thread Affinity is important in order to prevent the scheduler from migrating the threads for balancing issues. Hyperthreading is enabled only when all physical cores of the platform have been used.

- For the following results, the duration of each benchmark is 5 seconds. During this period, each thread performs randomly chosen operations, which are either *insert(key)*, *lookup(key)* or *deleteMin()*. The *key* of each operation is a random integer from 1 to double the initial size of elements at the structure. Between two consecutive operations, there is a time interval of 25 cycles, during which threads sleep. The evaluation is based on the **throughput** of the implementations. For the sake of correctness, each benchmark was executed 3 times, and the mean value of throughput was calculated.

- We experimented with different configurations, varying the size of the priority queue from small to larger data structures, using different numbers of threads, and different workloads, from read-intensive to write-intensive.

- At each benchmark, the data structure is initialized by a single thread at the beginning of the execution.

- We compiled our implementations using GCC 4.9.2 at sandman platform and GCC 5.3.0 at numascale platform, with -O3 optimization flag enabled.

4.2.2 Implementation details

Before presenting the results of our experiments, we should delve into some details of FFWD and NUDDLE implementations. As we have discussed at Chapter 3, communication between clients and servers is achieved by reading and writing on shared cache lines. In order to reduce the cache invalidations, each server responds to more than one client with the same cache line. The size of cache line determines the amount of clients that will share the same response. In our case, on both NUMA platforms, the size of a cache line is 64 bytes. The response block consists of N 8 bytes - slots, where N is the number of clients that share the same response line, and a 8-byte flag variable. As a result, the maximum number of threads that can share the same response block, is **7**, when the length of the cache line is 64 bytes.

Each thread is created with a unique ID, which specifies the core that the thread will be pinned to. At NUDDLE algorithm, when a thread is a server, it is responsible for one or more numa nodes, according to its ID. Respectively, when a thread is a client, its ID (i.e. the core that is pinned to) provides information about the server that will process its requests and the slot it spins on at the response cache line.

Each server, except from processing client's requests, makes its own operations at the queue. More specifically, after examining all groups of clients, a server produces a random key and accesses the data structure. At the following diagrams, both at FFWD and NUDDLE, each server responds simultaneously to 7 client threads. Both at sandman

and numascale, each numa node consists of 8 cores. Responding to 7 clients per numa-node implies that one core will remain idle. For the sake of fairness of our results, all PQ implementations follow the same pattern regarding the thread placement.

4.2.3 Evaluation on sandman

Each socket on sandman consists of 8 physical cores and 16 hardware threads. For the NUDDLE experiments, we assigned 8 threads of the same numa node to operate as servers, each one pinned to a different core. We have experimented with various numbers of threads as we can see from figures 4.4 and 4.5. Let us name the four sockets of sandman, as s0,s1,s2,s3. Server threads are pinned at the cores of s0, both at ffwd and nuddle. Now let N be the number of threads in our benchmarks. Then, at FFWD implementation, we would have 1 server, pinned at the first physical core of s0 and N-1 clients. At NUDDLE, if $N \leq 8$, we would have N servers pinned at the first N cores of s0, or if $N > 8$, there are 8 servers and N-8 clients. More specifically:

- when $N \leq 8 \Rightarrow$ N threads are pinned to the cores of s0 (no hyperthreading).
- when 8 < N $\leq 29 \Rightarrow$ 8 threads are pinned to s0, the next 7 are pinned to s1, next 7 to s2, and finally next 7 to s3.
- when 29 < N $\leq 57 \Rightarrow$ the first 29 threads are distributed as described above and for the next 28 threads, hyperthreading is used, placing 7 threads to each of the 4 sockets.
- when $N > 57 \Rightarrow$ more than two threads are pinned to some (or all) physical cores of the platform, i.e. creating the effect of *oversubscription*, following the pattern described above.

Some **conclusions** that we can assume by the examination of figures 4.4 and 4.5 are:

- **Ascylib's concurrent PQ** implementations scale adequately when the contention between threads is low. More specifically, at read-only benchmarks, regardless the size of the queue, their performance is increasing, when more and more working threads access the data structure. The same behavior is also observed when the number of elements at the structure is relatively large (1048576 and 2097152 keys).
- **Ffwd** implementation's throughput is, as expected, constant with the number of threads and limited by the single server's performance. It outpefroms ascylib's concurrent implementations only at very-high-contention cases, i.e. when the size of the structure is small (128 or 1024 keys), the benchmarks include write operations (insert-deleteMin) and the number of threads exceed the cores of a single numa-node.

- **Nuddle** algorithm's throughput is increasing when the number of threads is less than or equal to 8, i.e. the number of servers. Afterwards, as more and more client threads are added, the performance remains constant, as the servers are the only who access the data structure. When the parallelism is high, i.e. at read-only workloads, this limitation leads the asclylib's implementations to outpeform NUDDLE. On the contrary, as NUDDLE eliminates the numa effect, it performs the best among the others when the contention between threads is high. High contention appears at small data structures (128/1024 keys) and at larger queues with a higher ratio of update operations.

- Oversubscription seems to affect ASCYLIB's concurrent implementations in some cases (structure relatively large and update ratio $\neq 0$) while ffwd and nuddle maintain their constant performance.

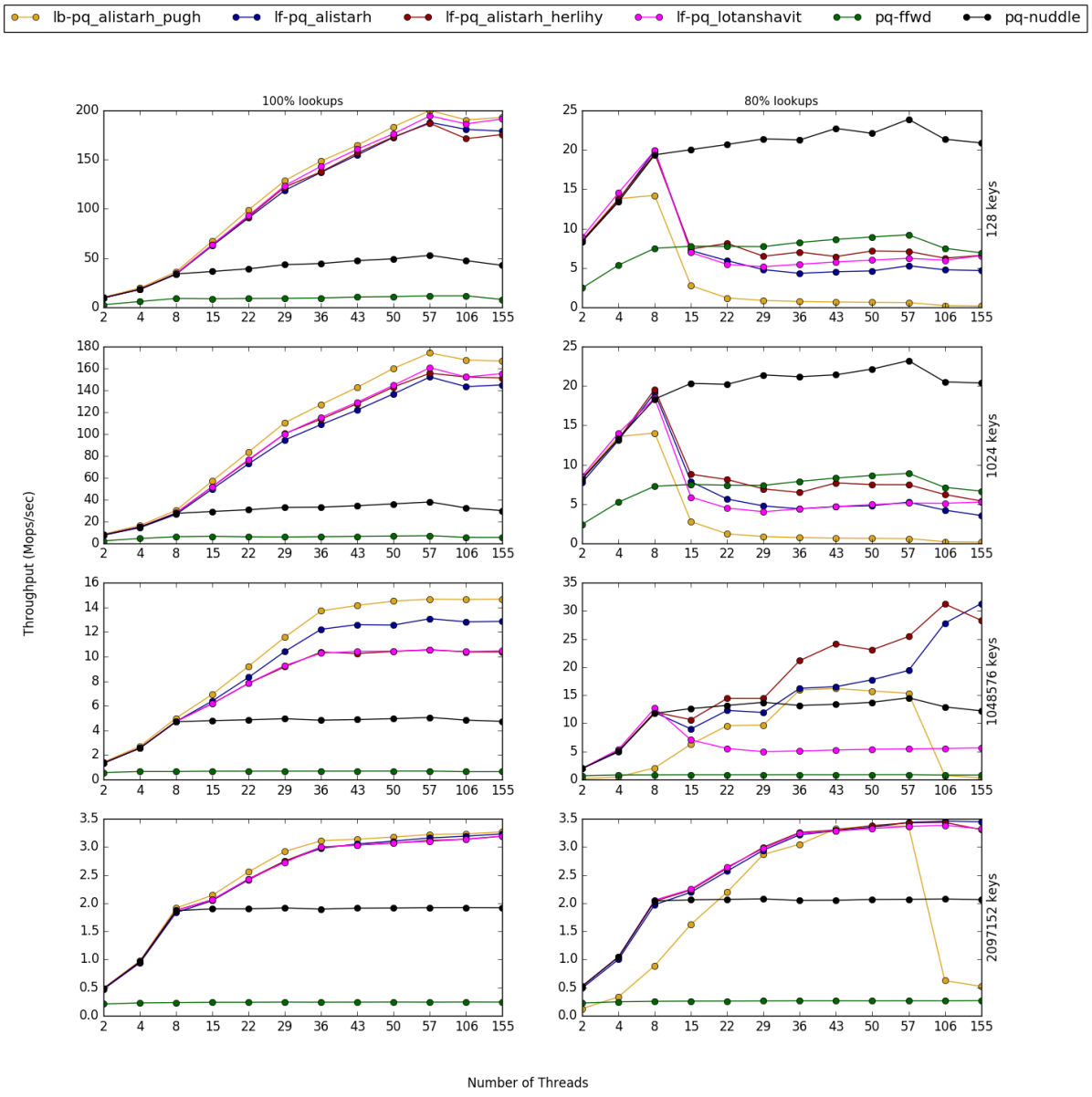


Figure 4.4: Evaluation of Priority Queue implementations on sandman. X-axis represents the number of working threads while Y-axis the algorithms' throughput measured in millions of operations per second. The rows represent different PQ sizes and the columns different workloads.

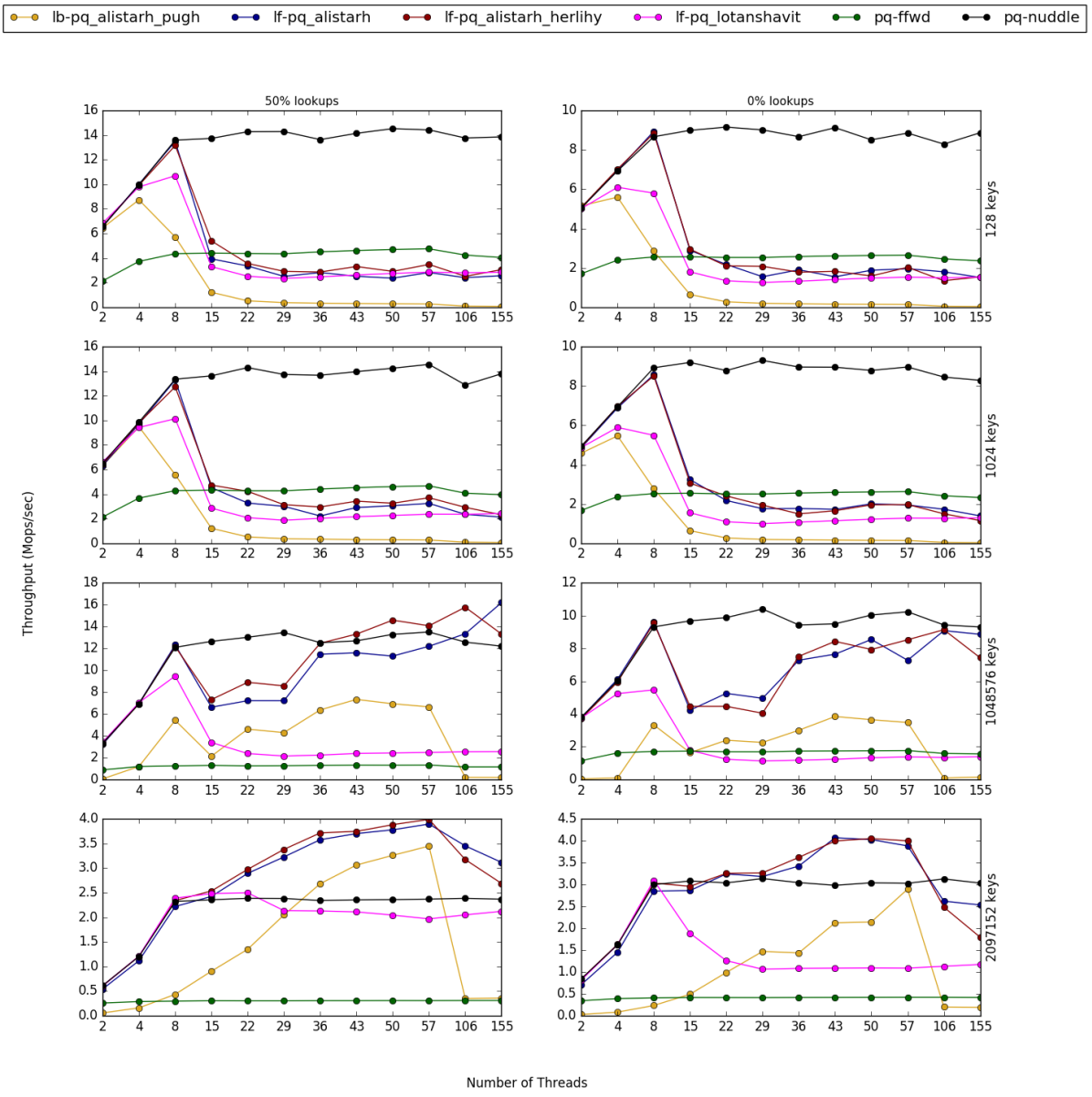


Figure 4.5: Evaluation of Priority Queue implementations on sandman. The first column represents a worload of 50% lookup, 25% insert and 25% deleteMin operations, while the second column represents an update-only benchmark with 50% insert and 50% deleteMin.

4.2.4 Evaluation on numascale

As we have discussed, numascale platform consists of 6 physical servers, each of them containing 3 AMD processors with 2 numa nodes of 4 physical cores, resulting at 24 cores, i.e. 24 threads at each physical server. We experimented varying the number of threads operating as NUDDLE servers from 4 (one numa-node), and 8 (one processor) to 24 (one physical server). We concluded that the best results are achieved using 24 NUDDLE servers, as we can not only reduce remote memory accesses but also achieve higher parallelism. Figures 4.6 and 4.7 display some of our experiments.

Thread affinity is used at this platform too. If N is the number of working threads in our benchmarks, and ps0, ps1, ps2, ps3, ps4, ps5 the physical servers of numascale then:

- if $N \leq 24$, then all threads are pinned to the 6 numa nodes of ps0. At the case of NUDDLE these threads operate all as servers.
- if $24 < N \leq 129$, then the first 24 threads are pinned to ps0, and the rest N-24 threads are distributed to the rest physical servers. We use 7 out of 8 cores of each processor, resulting in placing 21 threads at each of ps1,ps2,ps3,ps4,ps5. As a result, from the remaining 24 threads, the first 21 will be placed at ps1, the next 21 at ps2, etc.
- if $N > 129$, then the first 129 will be pinned as described above. The rest N-129 threads (*oversubscription*) will be distributed to all physical servers except from ps0: 21 threads at ps1, the next 21 threads at ps2 etc.

From figures 4.6 and 4.7 we can conclude that:

- As expected, **NUDDLE**'s throughput is equal to Alistarh-Herlihy's implementation when the number of threads is lower than 24, i.e. no clients exist, and afterwards it remains constant.
- **Ascylib**'s concurrent implementations perform here differently than on sandman. More specifically, at benchmarks with small size, they scale efficiently at read-only workload, while when update operations are executed, they scale until 8 threads and then their performance degrades. When the size of the structure is 1048576 elements, the throughput of lf-pq_lotanshavit degrades with the number of threads, while the other implementations scale up to 24 threads. This behavior is observed at the larger data structure as well. When the number of threads exceeds the first physical server, throughput is either constant or degrades.

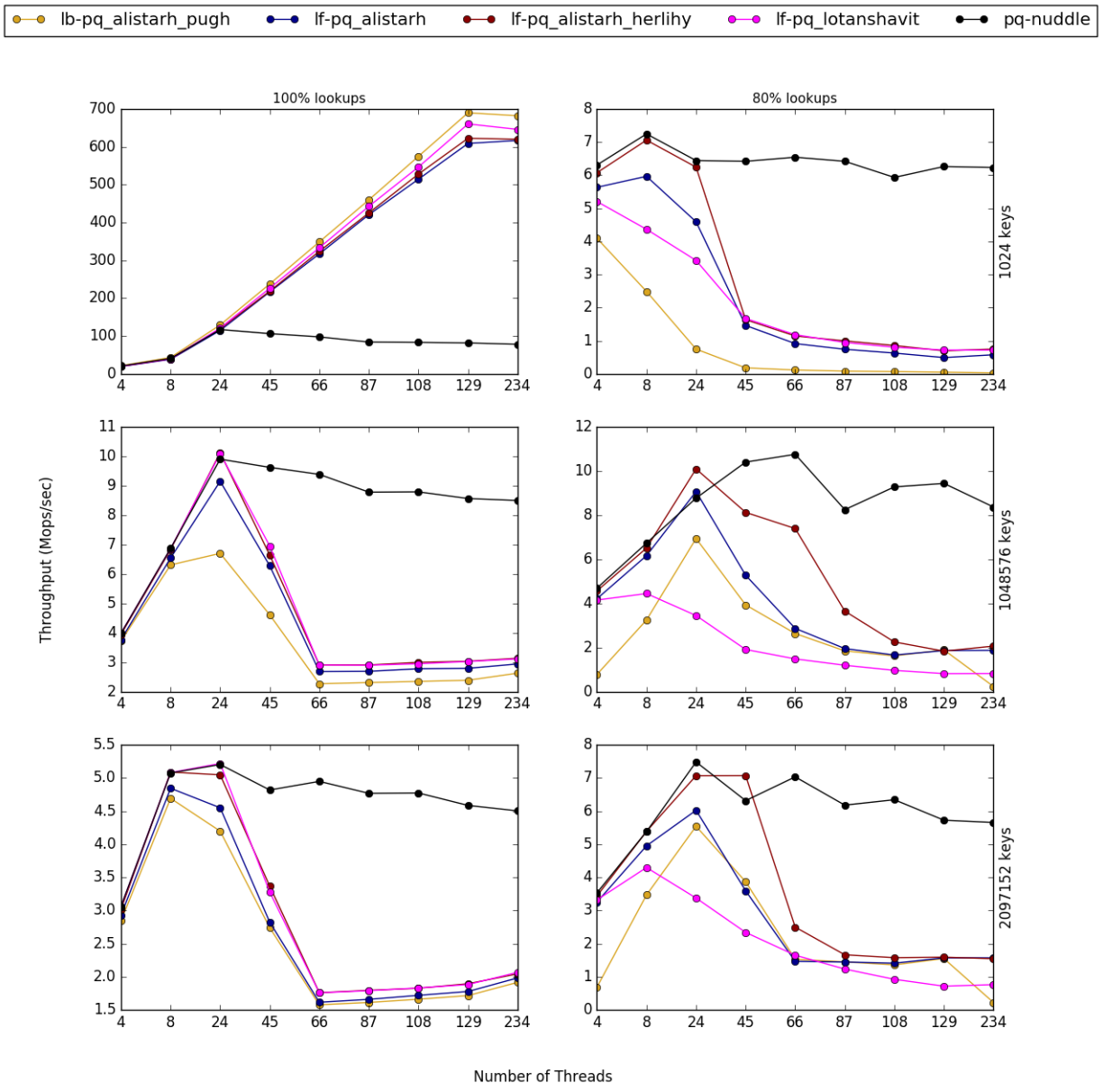


Figure 4.6: Evaluation of ascylib's concurrent priority queues and nuddle on *numascale* platform. The rows represent different sizes and the columns different workloads. The ratios of insert and deleteMin operations are equal.

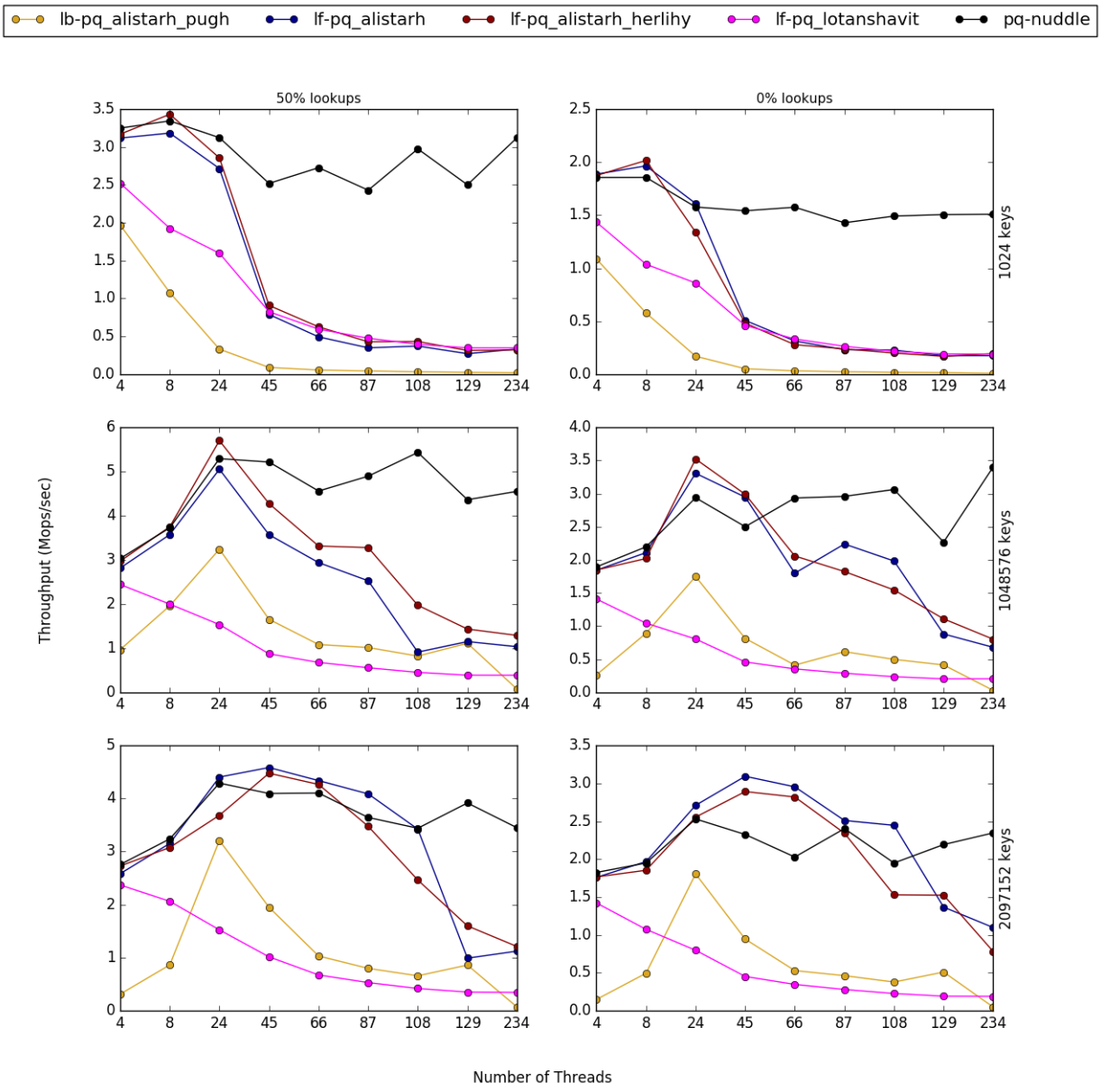


Figure 4.7: Evaluation of ascylib's concurrent pq and nuddle on numascale, for three different queue's sizes and two workloads. The ratios of insert and deleteMin operations are equal.

4.3 The need for a smart Priority Queue

Results presented at the previous section show that there is a **tradeoff** when designing a concurrent priority queue. This tradeoff results from the high parallelism provided by numa oblivious implementations and the data locality that characterizes numa aware implementations. Remote memory accesses and cache lines invalidations lead to degradation at the performance of high parallel PQ algorithms, especially at update-intensive benchmarks when the contention between the threads is heavy. On the other hand, numa-aware algorithms aim to reduce the amount of remote memory accesses and exploit the locality of a single numa node, thus limiting the number of threads that can access the data structure concurrently, and consequently limiting parallelism.

We observed that the PQ algorithms did not perform the same at both platforms, making obvious the fact that the platform's architecture strongly affects the performance of an implementation. Our experiments also point, that even at the same platform there are cases that numa-oblivious algorithms outperform FFWD and NUDDLE, and cases that numa aware implementations are more efficient and are benefited by the data locality they provide. As a result, we realized that there is not a state-of-the-art concurrent priority queue which outperforms all other implementations at any circumstances (size of the structure, workload etc.) at a NUMA system. The aforementioned tradeoff implies that when we gain parallelism, we lose in memory locality and vice versa. The conditions under which an application is executed, determine the kind of algorithm required, whether highly concurrent numa oblivious or a node local numa aware.

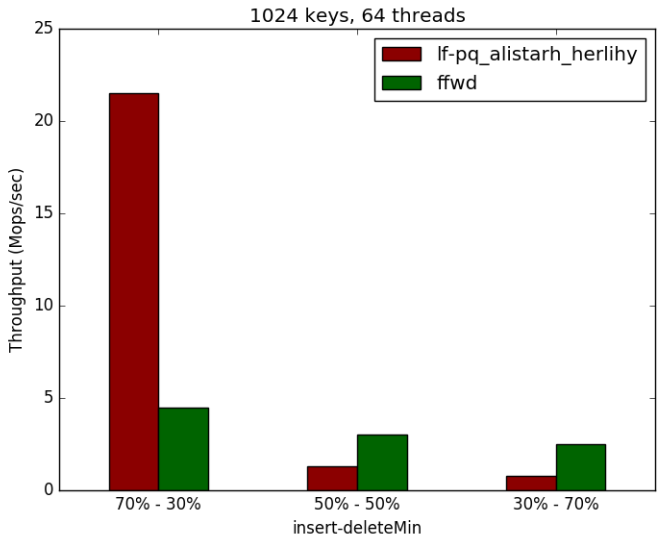


Figure 4.8: Evaluation on sandman of Alistarh-Herlihy spraylist and ffwd at a priority queue with 1024 elements. 64 threads generate a random key from the interval [1...2048] and perform either insert(key) or deleteMin. It is obvious that none of the implementations performs the best at all workloads.

Figure 4.8 compares two state-of-the-art priority queue algorithms, Alistarh-Herlihy's and ffwd and proves our observation: when the ratio of *insert(key)* operation is large, multiple threads can access concurrently various parts of the queue, the parallelism is high and the lock-free concurrent implementation is more performant. On the contrary, the *deleteMin* operations cause many threads to content for a small amount of memory locations. As a result, when the deleteMin ratio increases, the performance of Alistarh-Herlihy's pq degrades and ffwd outperforms it.

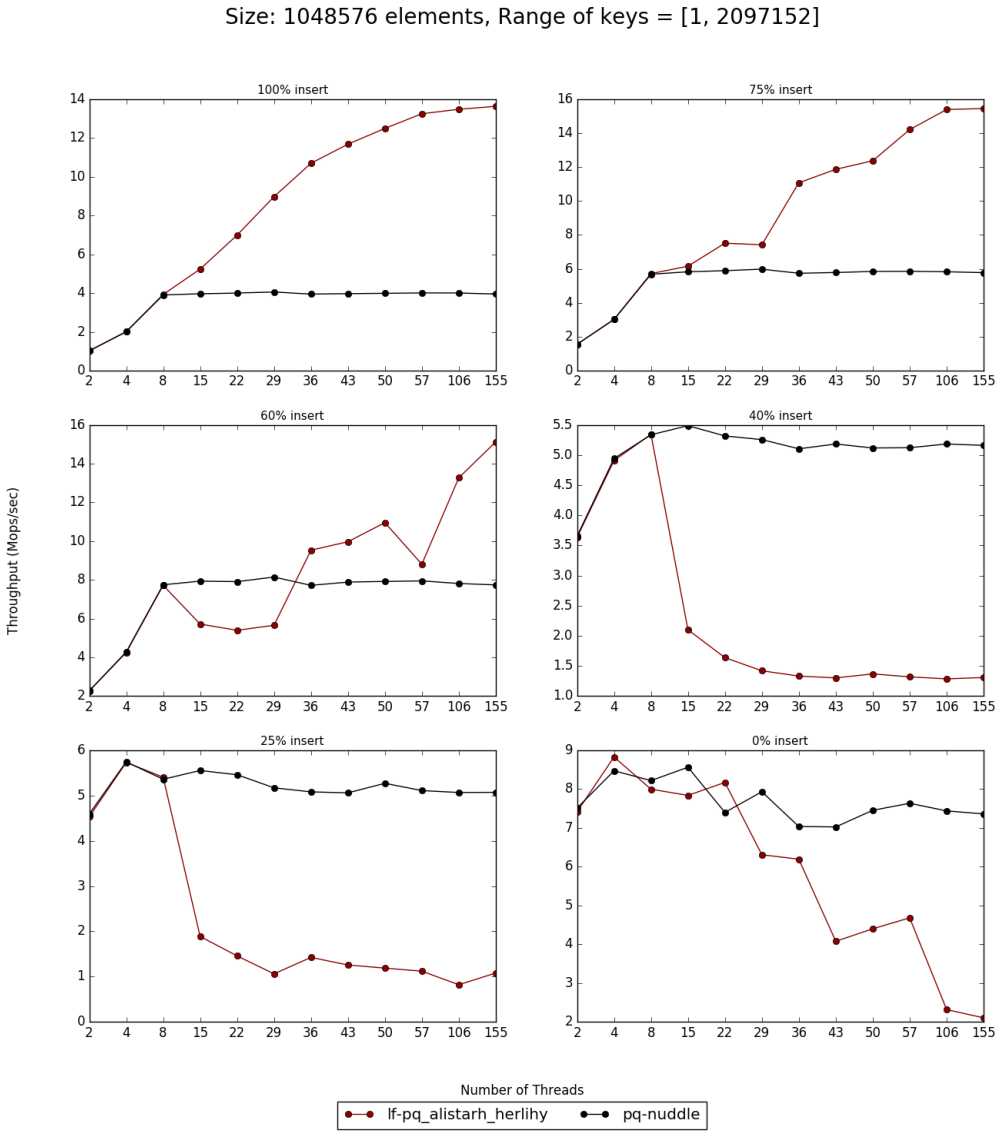


Figure 4.9: Evaluation on sandman of Alistarh-Herlihy's spraylist and nuddle. The size of the structure as well as the range of the keys remain constant. The benchmarks are all update-intensive: if x% is insert operations then (100-x)% is deleteMin operations.

Figure 4.9 illustrates the effect of the workload distribution, i.e. insert-deleteMin ratios at the performance of Alistarh-Herlihy's priority queue and nuddle algorithm. When the percentage of insert operations is high, the lock-free numa-oblivious implementation is a better choice, while at higher percentages of deleteMin, the numa-aware implementation achieves higher throughput.

Therefore, whether an implementation is efficient or not is judged under certain circumstances. A criterion that can affect the performance of an implementation at non-uniform architectures is the workload, i.e. the kind of operations executed at the shared data structure. Figure 4.10 points another factor that, according to our observations, can affect the efficiency of an algorithm: the range of the keys inserted at the data structure. We compared Alistarh-Herlihy's pq with nuddle, varying this range from 2000 to 100000 and we noticed that the relation between these two implementation's throughput varies as well.

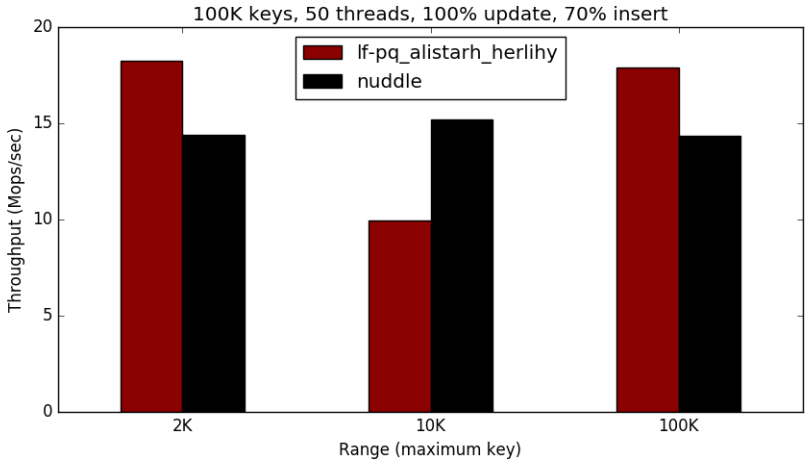


Figure 4.10: Evaluation on sandman of Alistarh-Herlihy's pq and nuddle. The queue contains 100000 elements, while the benchmark is update-intensive with 70% insert and 30% delete. The range of the keys generated at each iteration varies from 2K to 100K.

Furthermore, figure 4.11 clearly points that even if all other conditions (i.e. size of the structure, range of keys, workload) remain constant, the number of working threads can determine if a numa-oblivious or a numa-aware implementation should be used. Our implementation seems to be less efficient using a small amount of threads but outperforms the numa oblivious one when more and more threads are added at the execution.

The fact that the execution's configurations can have a huge impact at the performance of a concurrent priority queue at a non-uniform memory system, lead the PQ algorithms to present an undesirable behavior at many cases. This can affect the efficiency of parallel applications based on priority queues: a degradation at the performance of the baseline algorithm results to the degradation of the whole application and that may cost both time and money at many cases. To prevent this effect, we would like to design a concurrent priority queue that performs the best under all circumstances.

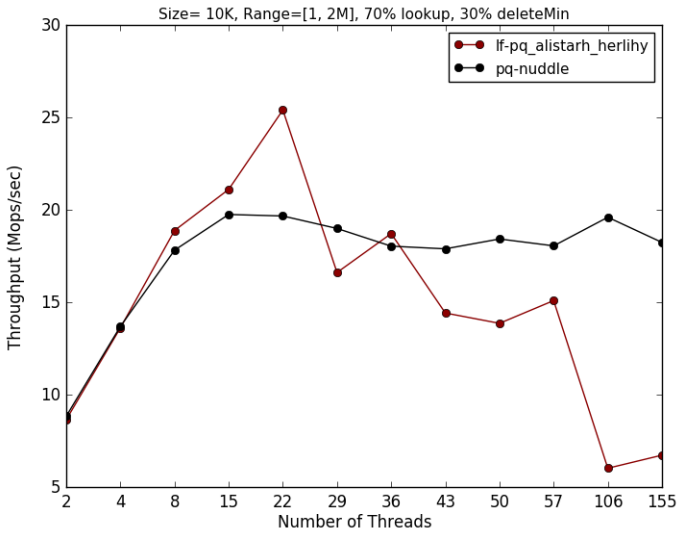


Figure 4.11: Comparison between Alistarh-Herlihy's spraylist and nuddle. All conditions of execution remained constant, except from the number of working threads. Spraylist performs the best when the number of threads is small, benefited by the concurrency of read operations. As more threads are added, deleteMin leads to high contention for a group of cache lines and nuddle is proved to be more efficient.

It became obvious that the tradeoff between high parallelism and data locality requires from the programmer to be aware of its application configurations, in order to decide whether a numa oblivious or a numa aware implementation would be ideal for a particular case. At this thesis we tried to make an approach to the designing of a *smart* data structure, i.e. a data structure that can achieve the best possible performance taking into account the conditions under which the application is executed, without extra effort from the programmer during the execution. The design of an *one-size-fits-all* data structure which will achieve the best possible performance regardless of the platform's specific architecture is exceptionally challenging, especially at non-uniform architectures. Therefore, we focused our attempts on *sandman* platform and evaluated our implementations there. However, the methodology presented at the next chapter can be utilized to create a smart data structure, e.g priority queue or stack at any NUMA platform.

Chapter 5

Designing an Adaptive Priority Queue Using Decision Trees

This chapter describes our approach towards the design of a self-aware priority queue. Before we delve into our work, we present some significant state-of-the-art previous approaches to auto-tuning concurrent data structures and synchronization techniques. Afterwards, we present our approach, justifying our decision to use machine learning, describing the whole process, difficulties that we met, and finally evaluating our implementations. Unfortunately, there is not an one-size-fits-all concurrent data structure. The performance of an algorithm is influenced by the conditions under which it is executed and by the machine's specific architecture. The previous chapter revealed the need for "intelligent" data structures that can be aware of the environment they execute on, and can be auto-tuned and adapt to dynamically changing conditions, in order to maintain their performance. Many factors affect the behavior of a concurrent data structure and trying to hand-tune an algorithm with regard to these factors in order to achieve high performance under all circumstances is challenging.

Machine learning provides tools that can be used to efficiently design self-aware, adaptive, self-optimizing systems, and therefore, researchers tend to deploy these tools for the development of self-aware computing. At this thesis we employed machine learning for the design of concurrent data structures that perform as best as possible on non-uniform architectures. We created a decision making mechanism that, regarding to the conditions of the execution, determines whether a numa-aware or a numa-oblivious algorithm should be selected. Afterwards, these algorithms and the decision mechanism were exploited for the design of a dynamic priority queue, that, when is needed, can proceed from one algorithm to another with inconsiderable overhead.

5.1 Related Work

At the beginning, we cite some significant work done at the field of adaptive and self-aware data structures and locking techniques:

- **Adaptive Locks: Combining Transactions and Locks for Efficient Concurrency**: Usui et. al. [34] proposes a technique that dynamically observes whether a critical section would be best executed using transactional memory or a mutex lock. There are the two *modes* of execution: mutex mode and transaction mode. Information such as the number of threads contending for the lock (when in mutex mode), the number of times a transaction aborts and has to retry (when in transaction mode), and the *transactional overhead* (how much slower is the critical section when in transaction mode), are computed dynamically and a highly effective *cost-benefit analysis* is executed. Keeping track of these statistics, the adaptive lock implementation can make a transition between the two modes introducing very low overhead and without effort from the programmer. The programmer has to declare the critical sections of the code that will be executed either with coarse grained lock or with software transactional memory (STM). When a thread wishes to access the critical section, the criteria described above are calculated and, regarding to the current mode, cost-benefit analysis determines whether the same mode is maintained or not. Adaptive locks combine benefits from both TM and mutex locking and can be extended and optimized to create concurrent data structures.

- **Smart Data Structures**: Eastep et. al. [14] proposes a new class of concurrent data structures that use online learning to automatically adapt and self-tune during the execution. The behavior of a data structure is affected by *knobs*, i.e. threasholds or other parameters of an algorithm. Eastep proposes a technique that uses an online learning engine to dynamically and automatically optimize knobs.

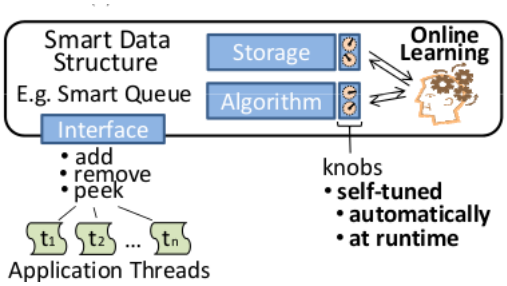


Figure 5.1: Smart Data Structures: Online learning mechanism used to self-tune knobs during execution

Figure 5.1 illustrates the basic idea of smart data structures. Knobs' self-tuning is achieved with *Reinforcement Learning*, and is driven by a reward signal regarding to throughput metric. All smart data structures share a common learning thread that runs a learning engine which jointly optimize their knobs. Online learning is used due to its ability to adapt to time-varying tradeoffs. Researchers et al. [14] presented a general methodology but focused on the optimization of *flat combining* algorithm (analyzed at chapter 3) outperforming static hand-tuned implementations.

- **Scalable Adaptive NUMA-aware Lock**: Zhang et. al. [37] proposes *SANL*, a scalable locking mechanism that can deliver high performance at non-uniform architectures under various contention levels by adaptively switching between in-place

locks and delegation locks. *In-place locks* require from threads to wait on shared variables before entering the critical section, and can cause many issues such as deadlocks and starvation. On the other hand, *delegation* implies that a server thread is responsible for executing a branch of operations, and although remote accesses to the shared data are minimized, the communication between clients and server may cause a large overhead, especially at NUMA system. SANL combines these two mechanisms of locking and optimizes the NUMA policy in delegation mode to better utilize the server and reduce remote clients' waiting times. More specifically, researchers et. al. [37] define contention levels, both node-local and global. The execution is partitioned into periods and at each period global and local contention are profiled with mathematical models, and using a voting algorithm the locking mode changes dynamically from delegation to in-place and vice versa.

5.2 Our Work

We continue describing the process we followed for the design of a smart priority queue for non-uniform architectures, that regarding its configurations, determines whether it will choose a numa-oblivious or a numa-aware implementation. We used the **Alistarh-Herlihy's spraylist** as the numa-oblivious implementation and our **NUDDLE** algorithm as numa-aware, since they share the same baseline concurrent skiplist algorithm, something very useful as we will see at the next subsection.

5.2.1 Why we used machine learning

The results presented at the previous chapter show that the performance of our known concurrent priority queue implementations is affected by many factors such as the size of the structure, the range of the keys generated and used for operations, the workload configurations, etc. Ideally, we would like to find a relation between all these execution parameters, using a statistical or a mathematical model, and implement a suitable function that regarding to the aforementioned model and the parameters given will determine the best algorithm for the execution. However, after many experiments we figured out that, in our case, the relation between the factors affecting the algorithms' efficiency is not straightforward and trying to create by hand even a simple and approximate model is very challenging. The diagrams below illustrate our attempts to figure out the effect of execution parameters on the spraylist's and nuddle's throughput.

Figure 5.2 presents the effect that oversubscription may have to the examined algorithms. As expected, nuddle's throughput is not affected as the number of server threads remain constant. On the contrary, at the first case, alistarh-herlihy's spraylist performs better with oversubscription and outperforms nuddle, while at the second case its throughput degrades and becomes worse than nuddle. We realized that we cannot easily predict the effect of oversubscription to spraylist's throughput and as a result we cannot easily determine in many cases the most performant algorithm.

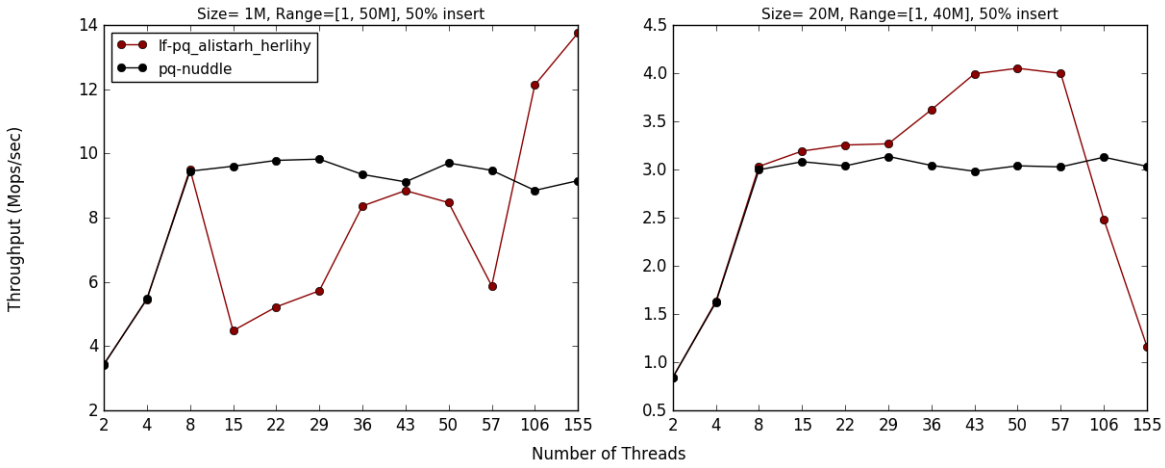


Figure 5.2: Evaluation of Alistarh-Herlihy and NUDDLE implementation with 50% insert and 50% delete operations. Oversubscription affects the performance of alistarh-herlihy.

Figure 5.3 illustrates a simple experiment where we varied only the initial size of the priority queue, keeping all other parameters constant. When the structure's initial size is increased from 100K to 1000000 elements, spraylist outperforms nuddle. However, a further increment at the queue's size reverses the relationship of the two algorithms, making it hard to predict the best algorithm when we change the size of the structure.

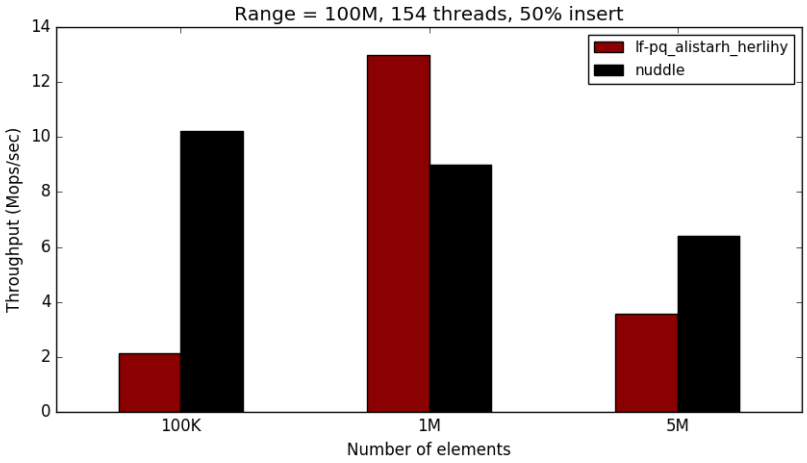


Figure 5.3: We evaluated spraylist and nuddle pq keeping all arguments constant except from the size of the queue. Although spraylist seems to perform better with 1M elements, when we increase more and more the size, nuddle outperforms it.

Another significant factor that affects the concurrent priority queues' performance is the range of the keys inserted at the structure. Figure 5.4 displays two experiments with different range values. For each experiment we evaluated three write-intensive workloads

varying the insert-deleteMin ratios. Neither at this case can we determine how range affects the implementations: at 50%-50% workload, a change in range reversed the relation between the two algorithms, while the other workloads remained unaffected. Finally, figure 5.5 displays two experiments which differ only to the number of threads. Again, the choice of the most performant implementation is not a trivial task.

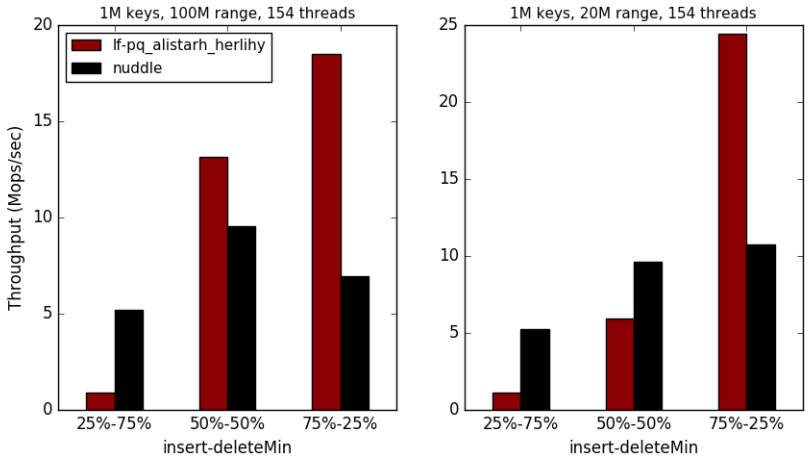


Figure 5.4: Spraylist's and nuddle's throughput at three write-intensive benchmarks with different range of keys.

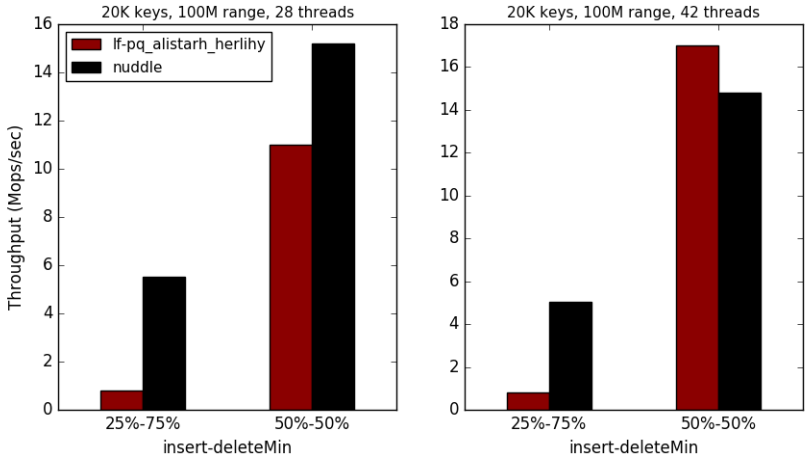


Figure 5.5: Throughput achieved by alistarh-herlihy's pq and nuddle for two workloads keeping all other parameters constant and varying the number of threads.

Therefore, the thresholds at which one implementation is more performant than the other are not clear. The size of the queue, the workload, the range of keys are only

some of the factors that determine whether a numa-oblivious or our numa-aware algorithm should be used. We have studied the effects of these factors and we have realized that the relationship between them and the algorithms' throughput is complex and cannot be straightforwardly computed. Consequently, we utilized machine learning tools to extract the requested threasholds of our decision making mechanism. The problem that we were called to solve here is a **classification problem**: we wish to determine which algorithm we should use regarding to the factors that affect the execution. As a result, our work was to create an adequate dataset, train our classifier, and by using parameter optimization and regularization methods, extract a suitable accurate predictive model. At the rest of the thesis we will delve into our classification problem, we will describe our methodology for building a decision making mechanism that selects between a numa-oblivious and a numa-aware implementation, and the way we utilized it to create a smart data structure and extend it to implement a dynamically changing, self-aware priority queue.

5.2.2 Machine Learning background

Machine learning is an extremely powerful tool and it can be used to efficiently solve decision problems. When the amount of data is large, when the solutions of a problem require lots of hand-tuning and long lists of rules, when statistical and mathematical approaches fail to find a solution, machine learning can be really helpful. As described in [18], there are many types of machine learning systems and it would be useful to categorize them with the following criteria:

- Whether or not they are trained with human supervision (supervised, unsupervised, semisupervised, reinforcement learning).
- Whether or not they learn and adapt their parameters *on the fly* (online versus batch learning).
- Whether they simply compare new data to old data points and make decisions, or instead they detect patterns in the training data and build a predictive model (instance-based versus model-based learning).

The process of a machine learning algorithm is illustrated at figure 5.6. The first step of this process is to collect a sufficient and representative amount of data. The dataset is perhaps the most important part of a machine learning process and consequently a lot of attention should be paid to this. After data preprocessing operations, the dataset is split to **training set** and **test set**. Training set will be used to create our model and tune its hyperparameters, while test set will be used to evaluate our work. This process can be iterative: the model created can be deployed, re-trained and re-tested until it sufficiently solves our problem.

Nowadays, machine learning is used to solve problems in a variety of fields. Briefly, problems that can be solved with machine learning techniques belong to two great categories: **classification** and **regression**. Let x be an input example and y the corresponding

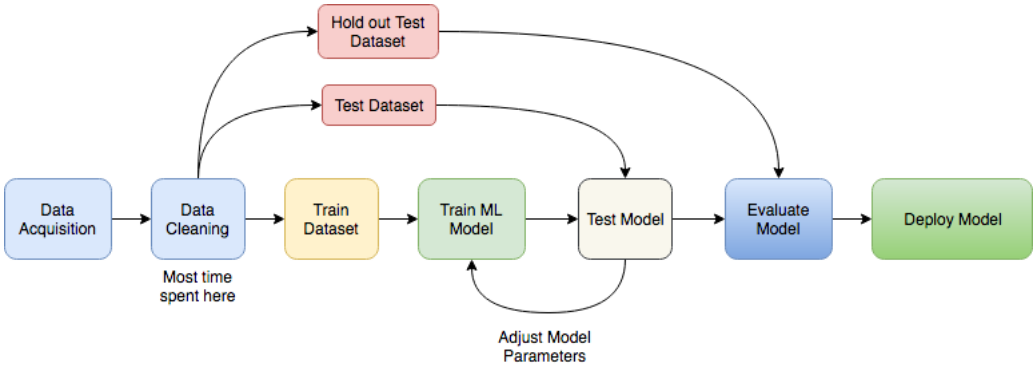


Figure 5.6: Machine Learning process overview

predicted output. At classification problems, y will be a discrete-categorical value, i.e. a discrete class label. On the other hand, if y is a real number (e.g an integer or a floating point value), then we have a regression problem. Some machine learning algorithms can be used for both classification and regression with small modifications, such as decision trees and artificial neural networks. Some algorithms cannot, or cannot easily be used for both problem types, such as linear regression for regression predictive modeling and logistic regression for classification predictive modeling.

In our case we were called to solve a classification problem. Regarding to the number of output labels, classification may be either binary or multi-class. Some of the most known classification algorithms are decision trees, logistic regression, naive bayes, k-nearest neighbors, linear support vector classifier, etc. For our predictive model we utilized decision trees, as they are simple, light-weighted and can be easily interpreted.

5.2.2.1 Decision Trees

A decision tree is a flowchart-like structure in which each internal node represents a *test* on an attribute (e.g. whether a coin flip comes up heads or tails), each branch represents the outcome of the test, and each leaf node represents a class label (decision taken after computing all attributes). The paths from root to leaf represent classification rules. Figure 5.7 illustrates the structure of a decision tree.

Decision Tree learning algorithms generate decision trees from the training data to solve **classification** and **regression** problem. Briefly, a decision tree algorithm works as follows:

1. Start with a training dataset which we will call S. Each item in the dataset is characterized by X attributes and belongs to 1 out of N classes.

2. Determine the *best* attribute in the dataset.

3. Split S into as many subsets as the possible values for the attribute determined at step 2.

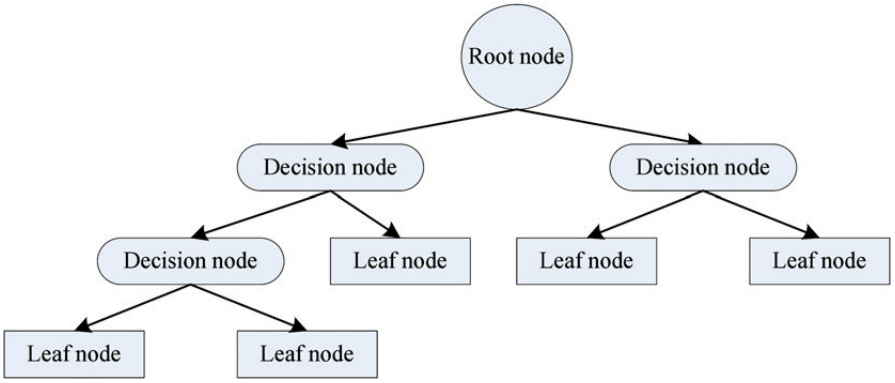


Figure 5.7: A decision tree consists of the root, decision nodes (questions for specific attributes) and leaf nodes (class labels). An element belongs to the respective class of the leaf node it ends up.

4. Create a decision tree node for each subset. Each node will be the root of a new, smaller decision tree.

5. Recursively generate new decision trees by using the subset of data created from step 3 until a stage is reached where you cannot classify the data further. Represent each class as leaf node.

In decision trees, the best attribute at each step is the one that gives us the largest **information gain**. Figure 5.8 illustrates two internal splits according to two different attributes of the dataset. As we have mentioned, each split leads to the creation of *k* subsets, where k is the possible values the respective attribute can take. In order to determine which split is the most suitable, i.e. which split provides more information gain, we will have to calculate the ***entropy*** of each subset.

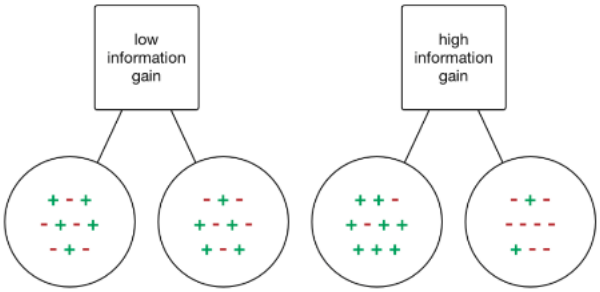


Figure 5.8: The left split has lower information gain, as the subsets created have almost equal numbers of '+' and '-' classes each. The right split is better since at the first subset class '+' dominates while at the left subset the majority of items belong to class '-'.

In machine learning entropy is the measure of homogeneity in the data. Its value ranges from 0 to 1. Its value is close to 0 if all examples belong to the same class and it is

close to 1 if there is almost equal split of the data into the different classes. The value of entropy for a set *S* is given by the following formula:

$$H(S) = -\sum_{j=1}^{c} p_j * \log p_j$$

where p_j represents the proportion of the data with j_{th} classification and c represents the different classes of the subset. Information Gain measures the reduction in entropy by "splitting" the data on a particular attribute. If *S* is a dataset (either the whole or a part of it if we are at an internal node), *A* the attribute we want to split on, *Values(A)* the possible values of A and *Sv* the subset of elements $\in S$ for which A has the value v. Then:

$$Gain(S, A) = H(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} * H(S_v)$$

where $H(S)$ is the entropy of set S, $|S|$ and $|S_v|$ the number of elements at S and Sv respectively and $H(S_v)$ the entropy of Sv. The above formula is calculated for each attribute A at each step of the algorithm. Each time, the goal is to maximize this information gain. The attribute which has the maximum information gain is selected as the parent node and successively data is split on the node. The procedure we described stops when all leaf nodes are pure or a maximal node depth is reached or splitting a node does not lead to an increase at the information gain.

Different algorithms use different metrics for measuring "best" attribute. Another splitting criterion is ***Gini Impurity***. Gini impurity is a measure of how often a randomly chosen element from the set would be incorrectly classified if it was randomly labeled according to the distribution of labels in the subset. It can be computed by summing the probability p_i of an item of class i times the probability $\sum_{k \neq i} p_k = 1 - p_i$ of a mistake in classifying that item. The lower the value of this metric, the more suitable the corresponding attribute is. When all elements at a node belong to the same class, gini impurity becomes 0.

If S is a dataset containing elements from J classes 1,2...,j and p_i the fraction of items labeled with class i in S, gini impurity is defined as:

$$I_G(S) = 1 - \sum_{i=1}^{j} p_i^2$$

In case of regression problems, where the output is a continuous quantity, attributes can be splitted into numerical intervals and similar steps are followed. Another splitting criterion, ***Variance reduction***, is often employed in regression cases, meaning that use of many other metrics would first require discretization before being applied. The variance reduction of a node N is defined as the total reduction of the variance of the target variable x due to the split at this node.

Decision trees mirror human decision making more closely than other machine learning approaches. They are easy to use and interpret and can be visualized providing a clear

picture of the classification rules. They are actually *white boxes* in the sense that the acquired knowledge can be expressed in a readable form, while KNN,SVM,NN are generally black boxes, i.e. one cannot read the acquired knowledge in a comprehensible way. Furthermore, they require little data preparation, they are able to handle both numerical and categorical data and perform really well with large datasets. On the other hand, a small change in the data can lead to a large change in the structure of a decision tree, making them unstable and non-robust. The problem of learning an optimal decision tree is known to be NP-complete. As a result, as we have mentioned above, practical decision-tree learning algorithms are based on heuristics (splitting criteria) such as the greedy algorithm where locally optimal decisions are made at each node. Such algorithms cannot surely guarantee to return the globally optimal decision tree, and even can create over-complex trees that do not generalize well from the training data (*overfitting effect*).

5.2.2.2 Classification

Before analyzing the procedure we followed for creating our decision making mechanism based on decision trees, it is helpful to mention some common challenges that occur on a classification problem and ways to overcome them, as well as to define some metrics that can be used to evaluate the quality of a classifier.

Overfitting vs Underfitting

A machine learning algorithm uses a dataset and creates and optimizes a model which is used to make predictions on a set of unknown data, named as test set. The performance of our model is judged with regard to the accuracy of the predictions it can make on unknown datasets. However, sometimes a model may give poor results. The poor performance of our model maybe because, the model is too simple to describe the target, or too complex to express the target.

Figure 5.9: Three classifiers for the same data. A model should not only caption the target but also generalize well on new data.

Figure 5.9 illustrates the problems of overfitting and underfitting. By looking at the graph on the left side we can predict that the line does not cover all the points. Such models

tend to cause underfitting of data (*high bias*). On the other hand, at the graph on the right side, the predicted line covers all the points. Except from the useful data, the predicted line covers also all points which are noise and outliers. Such models are characterized by *high variance* and they also tend to predict poor result due to their complexity.

Overfitting refers to a model that is an extremely accurate outline of the training data. It happens when a model learns the detail and noise in the training data to the extent that it negatively impacts the performance of the model on new data. This means that the noise or random fluctuations in the training data are picked up and learned as concepts by the model. The problem is that these concepts do not apply to new data and negatively impact the model's ability to generalize. Overfitting is more likely with nonparametric and nonlinear models that have more flexibility when learning a target function. Therefore, many nonparametric machine learning algorithms also include parameters or techniques to limit and constrain how much detail the model learns. Decision trees are nonparametric machine learning models, they are very flexible, and tend to adopt the form of the training data and as a result tend to overfit.

Techniques such as cross validation, dataset enrichment, features removal, early stopping, regularization, ensembling, etc. can be used to prevent a model from overfitting. In the case of decision trees the most common solution is ***pruning***, i.e. reduction in the number of nodes and the tree's depth in order to remove some of the detail it has picked up.

Underfitting refers to a model that can neither fit the training data nor generalize to new data. An underfitted machine learning model is not a suitable model as it will have poor performance on the training data. Underfitting is often not discussed as it is easy to detect given a good performance metric. When this effect is presented, a more complex model is required.

Classification metrics

In order to detect overfitting, underfitting and other problems of their models, data scientists rely on evaluation metrics. The most important of them are:

- **Confusion matrix:**

  Confusion matrix itself is not a performance measure, but almost all of the performance metrics we will analyze, are based on Confusion Matrix and the numbers inside it.

  Suppose we have a binary classification problem. There are two classes, positive (1) and negative (0). A machine learning model is trained at a dataset and evaluated at a test set. As we can see from figure 5.10, the predictions of the model at the test set can be illustrated using a *confusion matrix*. According to each element's actual and predicted class, it will be characterized as TP (true positive), FP (false positive), FN (false negative) and TN (true negative). In an ideal scenario, the number of false positive and false negative would be zero. However, real time models will not be 100% accurate most of the times.

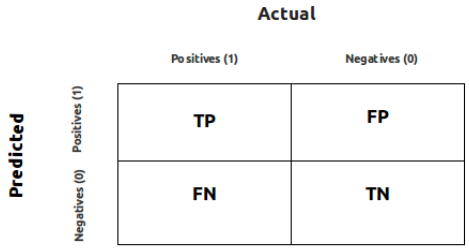


Figure 5.10: Confusion matrix of a binary classification problem

In a classification problem with N classes, the confusion matrix is a N*N array defined similarly to binary problems. Each row of the matrix represents the results of prediction for the corresponding class at that row, while each column represents the actual class. The diagonal cells show the number and percentage of correct classifications by the trained classifier, while the off diagonal cells represent the misclassified predictions.

- **Accuracy:** Accuracy in classification problems is the number of correct predictions made by the model over all predictions made. From figure 5.10 we can define:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

Accuracy is a good measure when the target variable classes in the data are **nearly balanced**. In cases of *skewed* datasets, i.e. in datasets where some classes are much more frequent than others, accuracy is biased to the most popular class and it should not be used as an evaluation metric.

- **Precision:** We can define precision using the confusion matrix as follows:

$$Precision = \frac{TP}{TP + FP}$$

Precision is a metric of the portion of positive predictions that were actually positive.

- **Recall or Sensitivity:** Precision is typically used along with recall metric, which shows the proportion of actual positives that was identified correctly:

$$Recall = \frac{TP}{TP + FN}$$

In a classification task, a precision score of 1.0 for a class C means that every item labeled as belonging to class C does indeed belong to class C (but says nothing about the number of items from class C that were not labeled correctly), whereas a recall of 1.0

means that every item from class C was labeled as belonging to class C (but says nothing about how many other items were incorrectly also labeled as belonging to class C). An ideal classifier is characterized by both high precision and high recall. However, there is an **inverse relationship** between these two metrics, where it is possible to increase one at the cost of reducing the other. It depends on the problem if the metric that we wish to satisfy is precision (when we want to ensure that our positive predictions are actually positive, i.e. reducing the number of false positives) or recall (when we wish to positively predict a high ratio out of actual positives i.e. minimizing false negatives)

- **F1 score:** A metric that combines the two aforementioned metrics is their harmonic mean, named as f1-score:

$$F1score = 2 * \frac{Precision * Recall}{Precision + Recall} = \frac{TP}{TP + \frac{FN+FP}{2}}$$

  It might be a better measure to use at datasets with uneven class distribution if we need to seek a balance between Precision and Recall.

- **AUC-ROC curve:** AUC - ROC curve is a performance measurement for classification problem at various thresholds settings. ROC is a probability curve and AUC represents degree or measure of separability, i.e. it displays the model's capability to distinguish between classes. Given a binary classification problem, the True Positive Rate, or the Recall that we have defined above is given by the type:

$$TPR = \frac{TP}{TP + FN}$$

  while the false positive rate is defined as:

$$FPR = 1 - TPR = \frac{FN}{TP + FN}$$

  The ROC curve is plotted with TPR against the FPR:

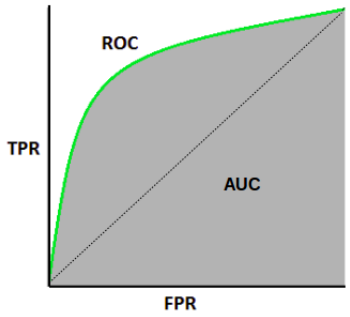


Figure 5.11: AUC-ROC curve

An excellent model has AUC near to the 1 which means it has good measure of separability. A poor model has AUC near to the 0 meaning that it cannot distinguish adequately between the classes.

- **Log Loss:** Logarithmic loss (related to cross-entropy) measures the performance of a classification model where the prediction input is a probability value between 0 and 1. The goal of our machine learning models is to minimize this value. A perfect model would have a log loss of 0. Log loss increases as the predicted probability diverges from the actual label. For a single sample with true label y_t in 0,1 (binary classification) and estimated probability y_p that y_t = 1, the log loss is

$$LogLoss = -(y_t * \log y_p + (1 - y_t) * \log (1 - y_p)$$

In multi-class classification, we take the sum of log loss values for each class prediction at this specific sample.

For further information about evaluation metrics on classification problems, reader can look up the corresponding literature. An informative view of evaluation metrics is presented at [32].

5.2.3 The smart priority queue

The first part of our work towards the design of a self-aware priority queue, was to formulate NUMA awareness as a classification problem and create a decision making mechanism to solve it. Giving a specific number of features, our model should determine which algorithm is the best between our numa-aware NUDDLE implementation and the numa-oblivious spraylist-based priority queue algorithm proposed by Alistarh [2]. In many cases, the throughput of the two algorithms is similar and therefore either nuddle or spraylist could be used. Consequently, we would like to create a classification model as shown in figure 5.12, that takes as input the values of five features: size, range, number of threads, lookup ratio, insert ratio, and predicts the label with the highest probability out of three classes: 0, 1 or 2. Each class determines which algorithm should be used: class 1 represents spraylist, class 2 represents nuddle while class 0 stands for ”tie”, meaning that either of these algorithms can be used providing the same throughput. We considered that both algorithms perform similarly when their throughput's absolute difference is lower or equal to 1.5, and we cannot determine which algorithm to select. Therefore these configurations belong to class 0, our ”neutral” class.

As we have mentioned above, the most important step of a machine learning process is the **training data accumulation**. Therefore, we created a set of 50000 different configurations by varying the values of our classification problem's features, trying to cover an adequate amount of test cases. Therefore, our dataset contained a huge variety of execution configurations, from small to big priority queues, from read-intensive to write-intensive workloads etc. Afterwards, we created two microbenchmarks for each

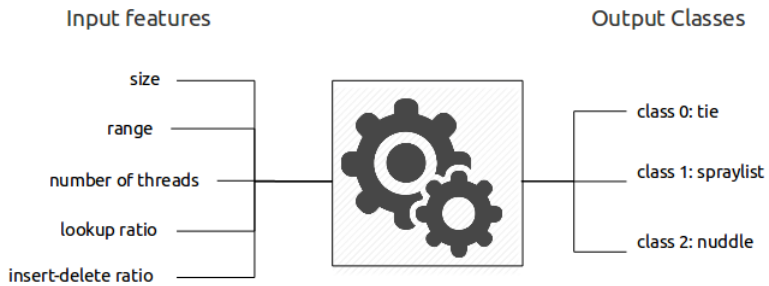


Figure 5.12: Modeling our classification problem using 5 input features and 3 output labels.

configuration, one for nuddle and one for spraylist and recorded their throughput. We executed our microbenchmarks as described at chapter 4, for 5 seconds each. The duration of execution could be regarded as an added feature but our experiments pointed that it does not affect intensively the performance of our algorithms and therefore we decided to keep it constant. After our benchmarks have been executed, we had to *label* our dataset: for each combination of features, we compared spraylist's and nuddle's throughput and we determined the class that this configuration belongs to, out of the three labels of our problem. More specifically, supposing that T_{sp} is the throughput of spraylist and T_n is the throughput of nuddle:

$$diff = T_{sp} - T_n$$

$$if\ -1.5 \leq diff \leq 1.5 => class\ 0$$
$$if\ diff > 1.5 => class\ 1$$
$$if\ diff < -1.5 => class\ 2$$

Consequently, we created a dataset of 50000 elements, each of them representing a different configuration, along with their labels. 80% of this dataset was utilized to train our classifier and the remaining 20% for evaluation. The next step was to find a model to train with this dataset in order to solve our **supervised** learning classification problem. We worked with decision trees, mainly because they are light-weighted and easily interpretable. More specifically, we employed **scikit-learn** ([25]), a python's framework for machine learning that contains decision trees implementations both for classification and regression problems and uses the CART algorithm for the training procedure. In general, the run time cost to construct a decision tree is $O(n_{samples} * n_{features} * \log n_{samples})$ where $n_{samples}$ is the number of samples used to build the classifier and $n_{features}$ is the number of features at the classification problem. The query time for such a classifier is $O(\log n_{samples})$.

At first we utilized a naive decision tree classifier with the default parameters and trained it with our dataset. Despite its adequate accuracy (around 90%) the depth of our tree was too large, increasing the danger for overfitting. In order to improve our model

and increase its accuracy at unknown data, we experimented with various decision tree parameters using 10-fold cross validation with random and grid search. The parameters with which we experimented are:

- **criterion**: The function to measure the quality of a split. Possible values are either *gini* or *entropy*, as we have discussed above.
- **max_depth**: The maximum depth of the tree.
- **min_samples_split**: The minimum number of samples required to split an internal node.
- **min_samples_leaf**: The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least min_samples_leaf training samples in each of the left and right branches.

We run cross-validation method evaluating each time our model with a different scoring function. At first we used accuracy as our scoring function but since our training set was imbalanced we experimented as well with precision, recall, f1-score (both micro and macro averages) and log loss. The best models found are presented at table 5.1. We evaluated each classifier presented below with the 20% of our dataset and calculated a set of metrics for each one of them. Furthermore, we measured our model's accuracy at a dataset of 11000 unknown entries. Each entry represented a different configuration and was labelled with the appropriate class. The aforementioned results are presented at table 5.2.

	Tree Parameters			
	min_samples_split	**criterion**	**max_depth**	**min_samples_leaf**
1	2	gini	none	1
2	10	entropy	13	1
3	8	entropy	13	5
4	10	entropy	13	3
5	2	entropy	13	5
6	8	entropy	14	3

Table 5.1: The best models generated by cross validation

	Test set 1					Test set 2
	Accuracy	**Precision**	**Recall**	**F1-score**	**Log Loss**	**Accuracy**
1	0.907	0.896	0.895	0.896	3.205	0.793
2	0.924	0.913	0.917	0.915	0.718	0.813
3	0.922	0.913	0.914	0.914	0.577	0.795
4	0.934	0.914	0.919	0.916	0.666	0.891
5	0.926	0.914	0.918	0.916	0.645	0.833
6	0.924	0.914	0.918	0.915	0.917	0.809

Table 5.2: Evaluation of the best models generated with cross validation

In order to validate the results produced by cross validation process we hand-tunned our model with the aforementioned parameters. Diagrams 5.13, 5.14, 5.15 and 5.16 illustrate the way that max_depth parameter affects the accuracy, precision, log loss and auc score of our model. We can see that when the max depth of our tree is between 11 and 15, all these metrics have a sufficient value. We also experimented with various values of min_samples_split, criterion and min_samples_leaf, as well as with many combinations of them, visualizing their effect on the aforementioned metrics.

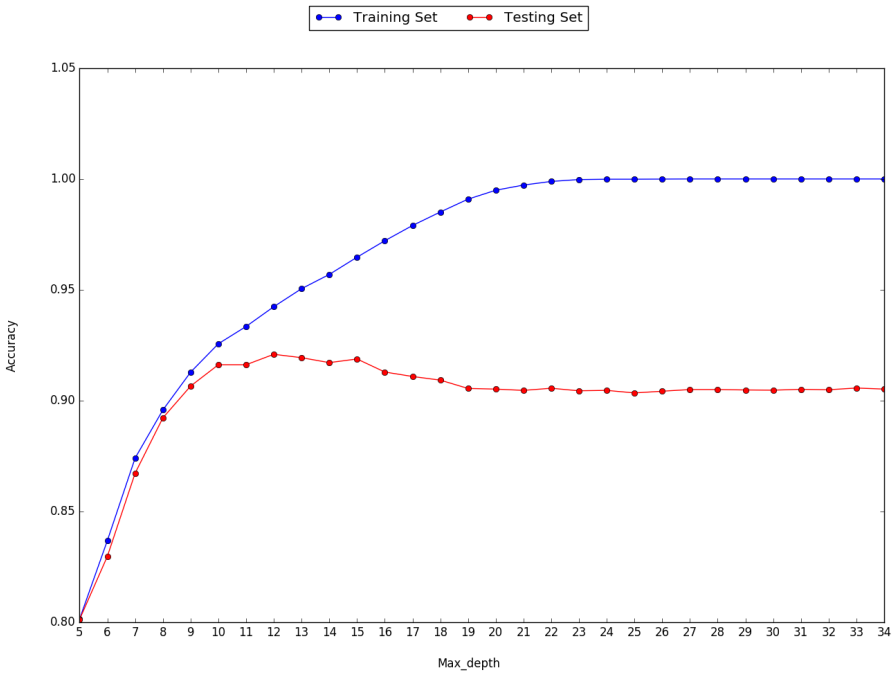


Figure 5.13: Average accuracy at training and test set with regard to max_depth parameter.

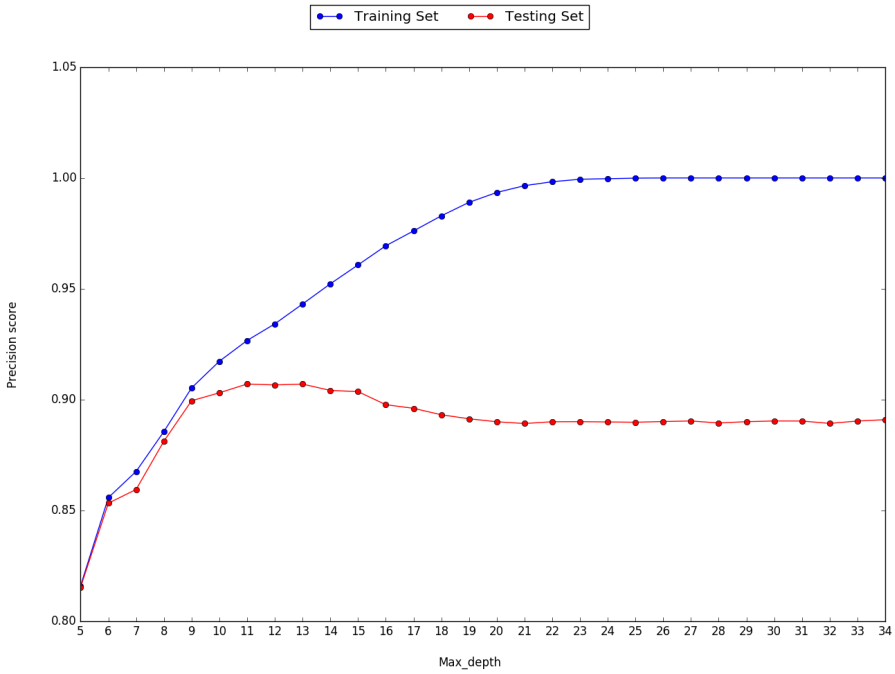


Figure 5.14: Macro-average precision at training and test set with regard to max_depth parameter.

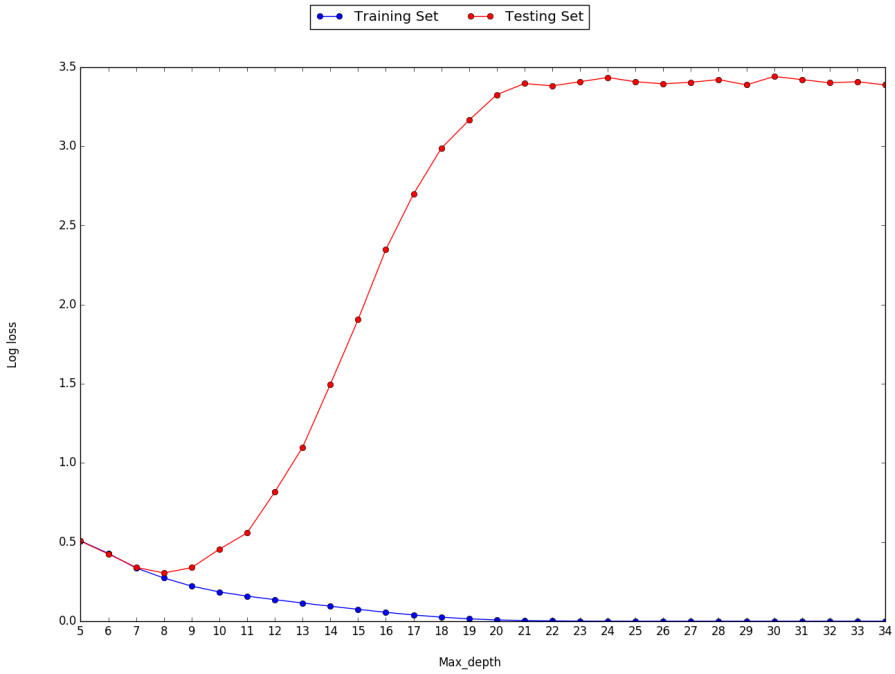


Figure 5.15: Logarithmic Loss at training and test set for different values of max_depth.

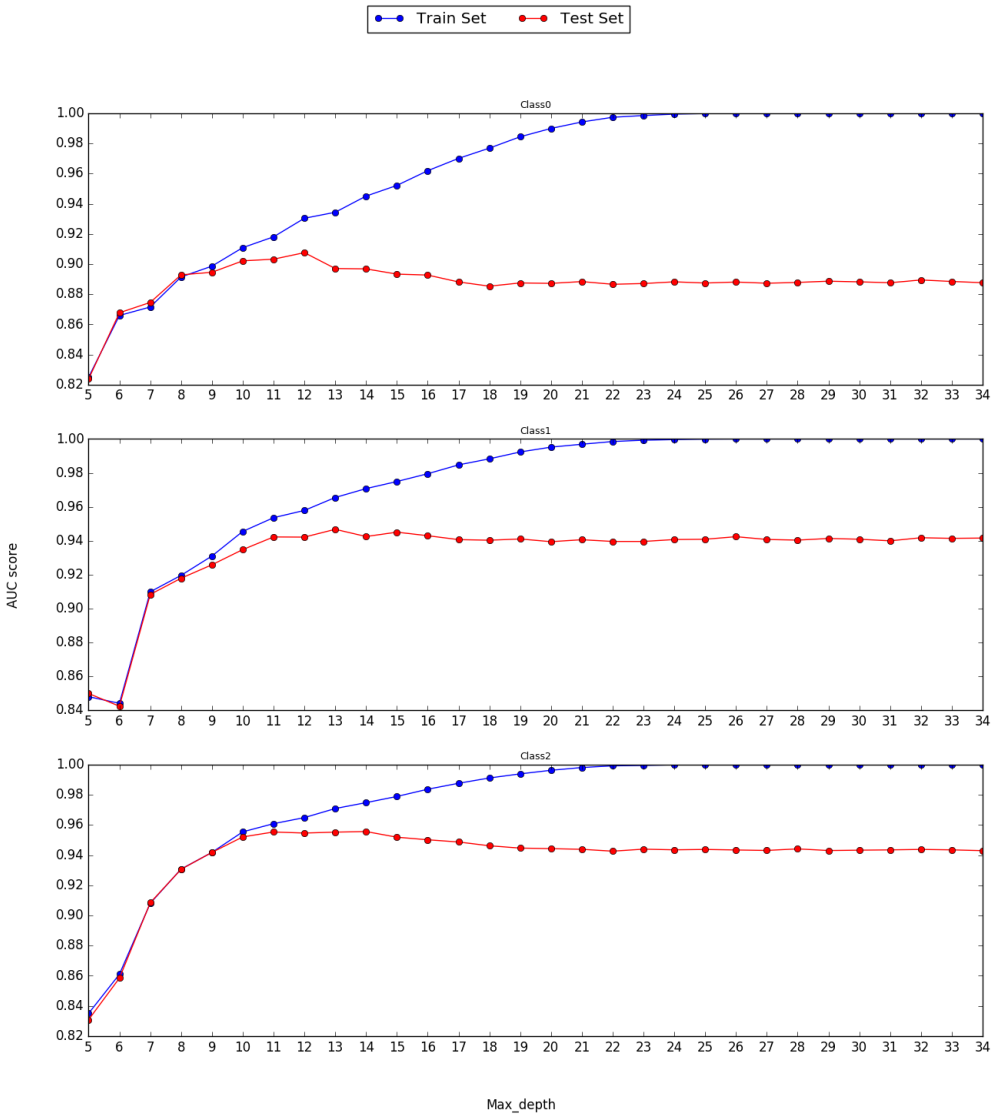


Figure 5.16: AUC score for each class for different values of max_depth.

Our final model was a classifier with the following parameters:

Parameter	Value
min_samples_split	10
max_depth	13
criterion	entropy
min_samples_leaf	3

Table 5.3: Final model after parameter tunning process.

Afterwards, we interpreted our decision tree classifier using a tree traversal algorithm and generated a C function that contained the rules of our model. Our smart priority queue takes as input the initial size, the keys range, the number of threads operating and the workload, calls this function and either executes nuddle or spraylist implementation. We evaluated the accuracy of our classifier and as a result the efficiency of our smart priority queue in a variety of benchmarks shown at figures 5.17 and 5.18. We tested each configuration varying the number of threads and we noticed that smart pq selects the best algorithm at each case and as a result achieves the highest possible throughput.

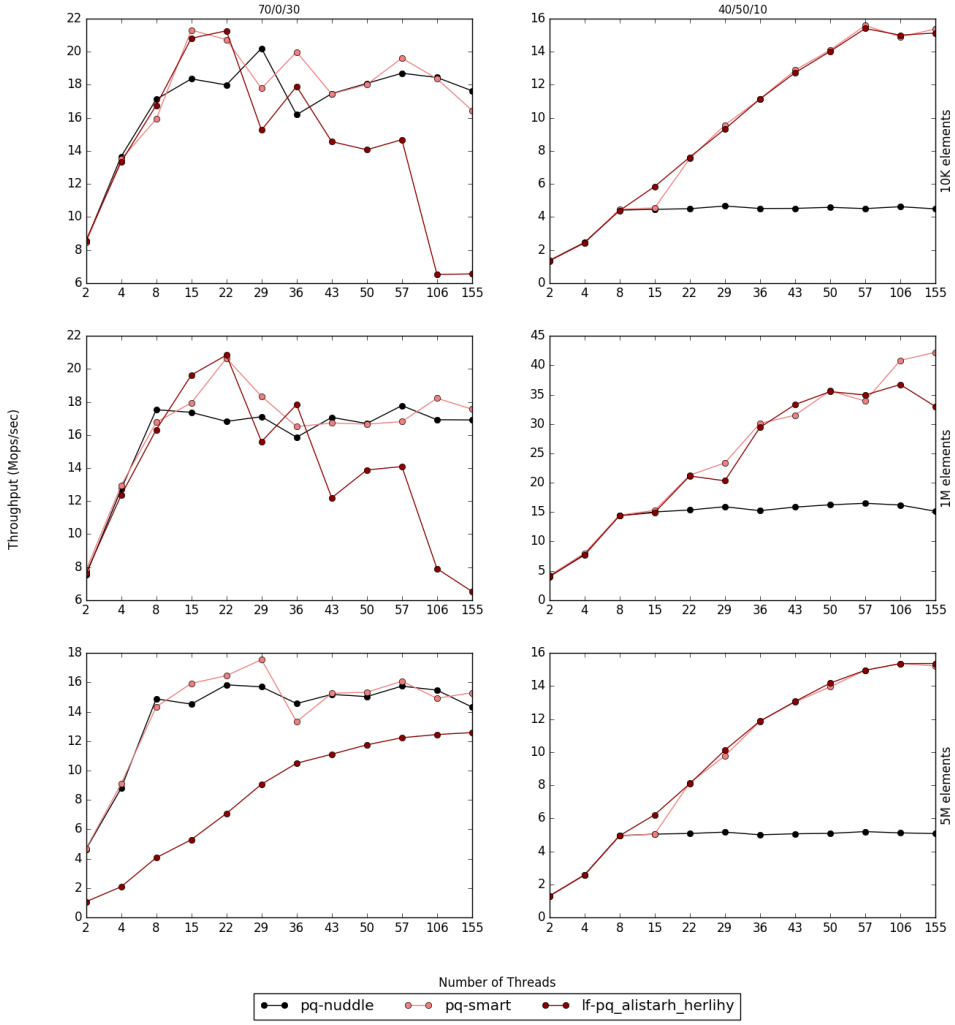


Figure 5.17: Spraylist, nuddle and smart priority queue evaluated on sandman. The rows represent different PQ sizes and the columns different workloads of lookup/insert/deleteMin operations. The smart pq selects each time the best implementation and reaches the best possible throughput.

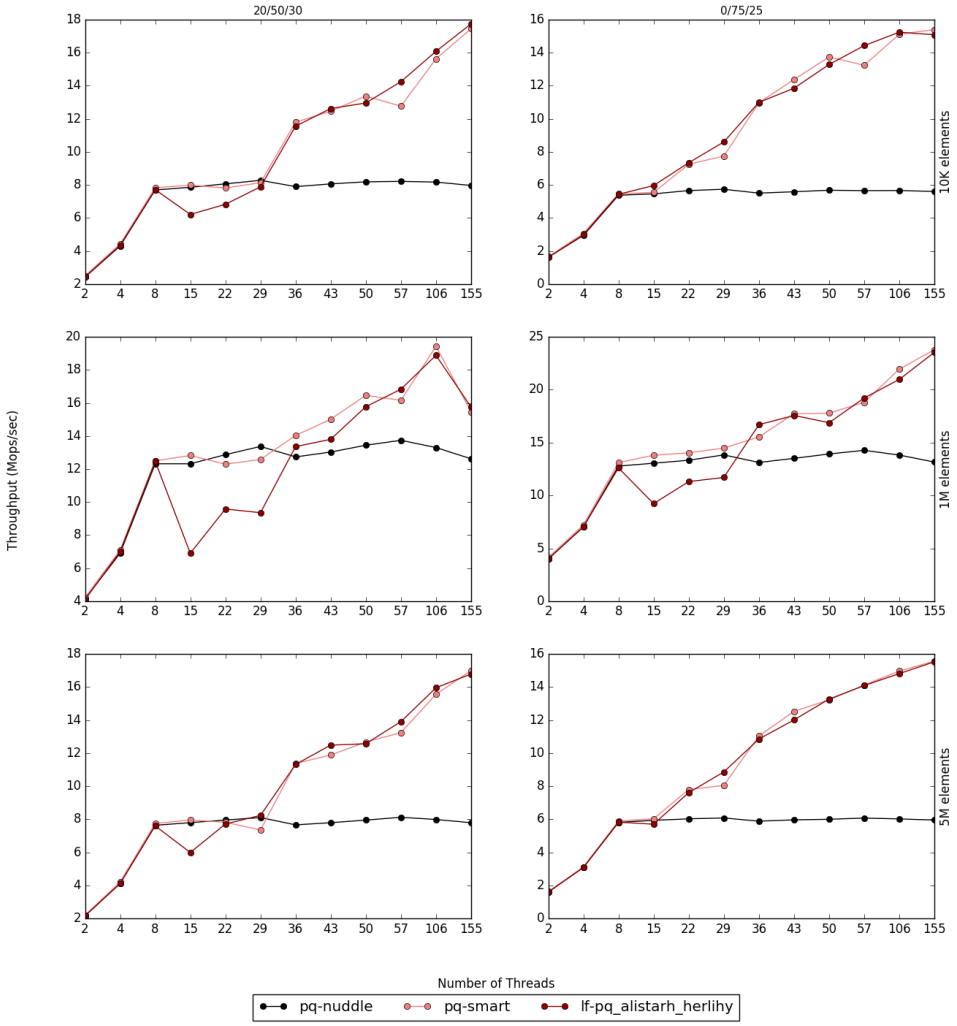


Figure 5.18: Evaluation of smart pq, spraylist and nuddle for two different workloads and three sizes. Smart PQ chooses the best implementation at each case.

5.2.4 Our smart priority queue in a dynamically changing environment

Finally, at this subsection we present our approach towards a self-aware priority queue. We tried to simulate a dynamically changing environment, i.e. an environment where the conditions of execution are changing. Real-world applications are such environments, since the operations requested at the data structure vary during the execution. In order to maintain high performance, researchers need to design algorithms that can detect configurations' changes and adapt to them.

We extended our work to design an adaptive priority queue that can operate on a dynamic environment. Figure 5.19 illustrates the idea of our adaptive priority queue.

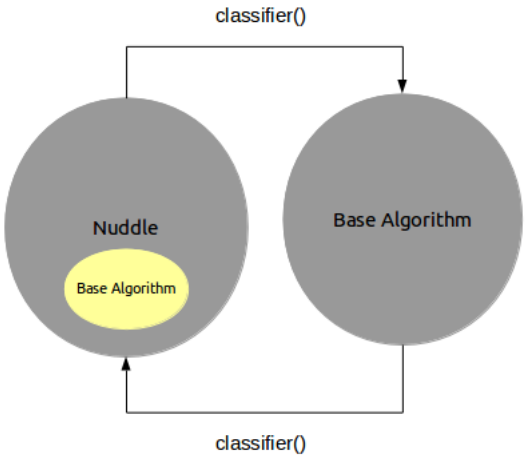


Figure 5.19: Our adaptive priority queue can move from nuddle to its baseline numa-oblivious algorithm and vice versa, with respect to the predicition of the decision tree classifier.

Our methodology is based on the idea that given a numa-oblivious data structure, spraylist in our case, one can use our NUDDLE framework as described at chapter 3 to create a numa-aware version. Afterwards, these two versions can be unified into one adaptive data structure, that each time it detects a change at the configurations, will call the decision tree classifier with the requested features and either remain at the same implementation or not. Therefore, in our case, all threads will execute either spraylist and directly access the shared priority queue, or NUDDLE and, thus, some of them will operate as clients and others will operate as servers.

The main advantage of our adaptive implementation is shown in figure 5.19 and it is basically that both our **numa-aware and the numa-oblivious algorithm are based on the same core operations**. This means, that even at transition periods, some threads are allowed to delegate their operations to servers, while others can directly access the data structure without violating corrrectness, as the same basis operations are executed. On the other hand, if these two implementations did not share the same underline algorithm, a synchronization point should be added, when a transition is needed, to ensure correctness. This synchronization point (i.e. a barrier) may cause an additional overhead and lead to performance degradation.

We evaluated our adaptive priority queue, nuddle and spraylist using a series of dynamic benchmarks, each one consisting of two or more periods. As a period, we define a time interval (25 sec in our experiments) at which the keys range, the workload and the number of threads operating at the structure remain constant. After a period, one or more of the aforementioned features change. Then, all threads of smart pq call the classifier function which determines if they need to alternate implementation or not. At figures 5.20

and 5.21 we have changed the workload at each period, at figure 5.22 we have changed the range of keys of each operation, while at figure 5.23 we have changed the number of working threads that access the data structure. We can see that our adaptive priority queue chooses each time the most performant algorithm achieving the highest available throughput.

A possible future evaluation of our work would be in an actually dynamic environment e.g. a database system or a Dijkstra algorithm, where the execution's configurations could be extracted on-the-fly. Our numa-aware framework, our classification mechanism and the methodology proposed for adaptive data structures can be utilized with any efficient numa-oblivious algorithm to design implementations that will have the highest performance under all circumstances.

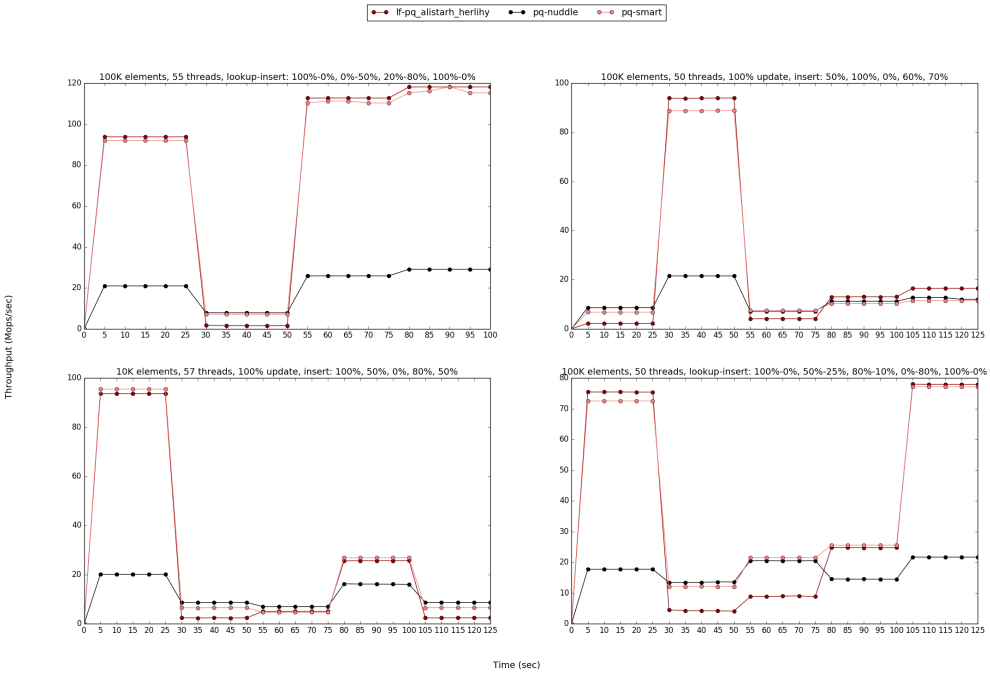


Figure 5.20: Nuddle, spraylist and smart implementation executing at a dynamically changing environment. Threads are running for 4 periods (first row) or 5 periods (second row). Each period lasts for 25 sec and has a specific configuration. At each period we vary the workload, i.e. the lookup-insert-deleteMin ratio.

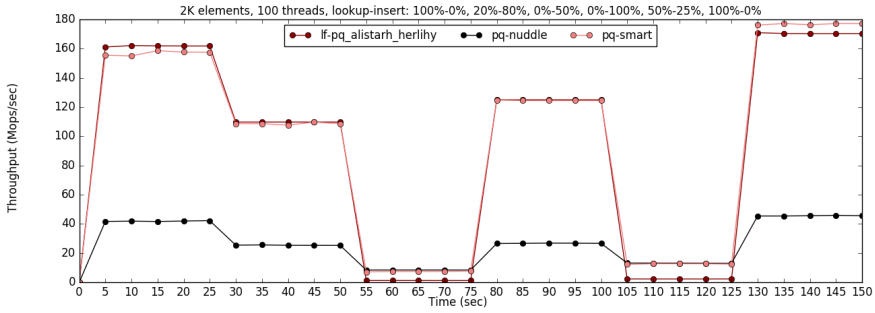


Figure 5.21: Nuddle, spraylist and smart implementation run for 6 periods, while each period lasts for 25 seconds. We change the workload at each period to simulate a dynamically changing environment and we can see that smart pq selects the most performant algorithm according at any point of time.

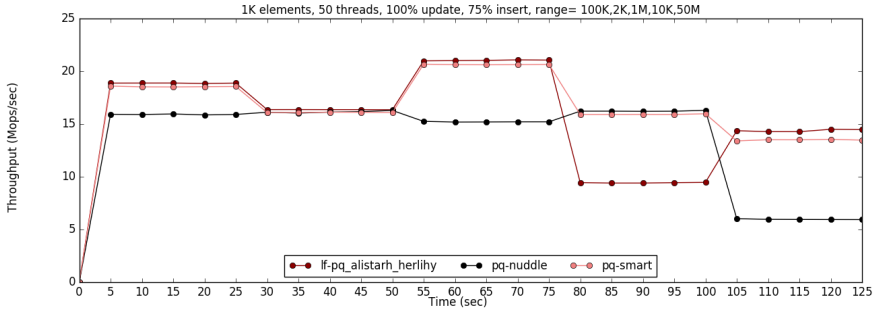


Figure 5.22: Nuddle, spraylist and smart implementation run for 125 seconds, with range of keys changing every 25 sec., while the number of threads and the workload remain constant.

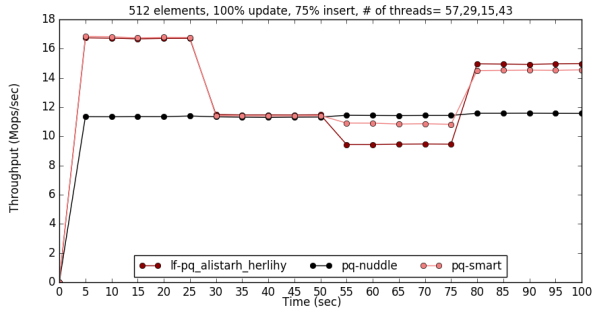


Figure 5.23: At this benchmark 57 threads are created but the number of them which make operations at the structure varies every 25 seconds. Again smart pq reaches the highest available throughput at each period.

Chapter 6

Conclusion and Future Work

At the beginning of this thesis, we evaluated some widely-used concurrent data structures from ASCYLIB library [10] at a system with a non-uniform memory architecture. We observed that data structures such as binary search trees and skip lists are not affected by the characteristics of NUMA systems, and present high scalability when more and more working threads access the shared data structure. On the other hand, we noticed that the performance of our known state-of-the-art priority queue implementations degrades when using a NUMA machine. Since priority queues are used in a variety of algorithms, we decided to evaluate further on these data structures, analyze the factors that lead to this performance degradation and try to overcome them.

A memory access in a NUMA system can either be node-local and be served quickly or be remote and more time consuming. Numa nodes, or *sockets* are connected with an interconnection network and all remote main memory and cache requests are served through this network. When the amount of remote accesses and the memory invalidation pace is high, the traffic at the interconnection network is intensified, increasing the latency of operations and thus degrading performance. This phenomenon is known as **numa effect** and it is the main reason why concurrent algorithms may not scale well on a NUMA system. As we have seen, priority queues are highly influenced by numa effect. Each element in the a priority queue is assigned with a specific key, which determines its priority. A thread that accesses a priority queue can basically search for a specific key, insert a new key-value pair, or delete the element with the maximum or minimum priority. The last operation, deleteMin (or deleteMax) causes threads to compete for a small amount of cache lines, increasing the number of memory invalidations and as a result causing the numa effect. When more and more threads wish to delete elements from the structure, this effect is intensified leading to degradation of performance. In order to overcome this constraint, relaxed algorithms have been proposed according to which, threads attempt to delete an element within a range of the k smallest (or largest) ones. However, our evaluation pointed that neither these implementations can scale adequately at a NUMA system.

Consequently, there is a need for numa-aware data structures. Towards this direction, researchers have proposed scalable implementations, either black-box (methodologies that can be utilized at any data structure) or specific for stacks, FIFO and priority queues, i.e.

structures at which are characterized by high contention points and numa effect is more intensive. Ideas such as memory replication [7, 9], operations elimination, delegation and combining [6, 5] are the main parts of these numa-aware implementations, which aim to fully exploit the single's numa node efficiency and limit the traffic at the interconnection network.

At this thesis we delved into the idea of delegation, according to which only a specific number of threads, called servers, can access the data structure and execute operations on behalf of the other threads (clients). We evaluated *fast fly-weighted delegation (FFWD)*, an implementation proposed in [29] with multiple clients and a single server which is the only one that accesses the structure (a priority queue in our case) and thus, there is no need for any synchronization. FFWD dedicates a specific cache line to each client for its requests and another cache line for server's responses to a group of clients that are located at the same numa node. Therefore, it manages to minimize the number of cache invalidations and the overhead caused by the server-clients communication. However, since only one thread accesses the data structure, threads' accesses are serialized (there is no parallelism) and the performance is limited to a single thread.

To overcome this limitation in performance, we extented FFWD by adding more servers. We proposed NUDDLE (Numa Node Delegation), a framework that can transform any numa-oblivious concurrent data structure into its corresponding numa-aware one. The servers have to be located at the same numa node and process clients' requests. As a result, data remain in a single's node main memory, thus reducing remote memory accesses and more threads operate on the shared structure, increasing the parallelism and leading to higher throughput. We exploited NUDDLE technique and spraylist algorithm, a relaxed skiplist-based concurrent priority queue proposed at [2], and created a numa aware priority queue. Although NUDDLE queue outperforms other implementations in write-intensive workloads and in cases when the contention between threads is high, performance is again limited by the number of servers. This led us to the realization that, when designing a priority queue for a NUMA system, there is actually a **tradeoff** between higher parallelism and memory locality.

After a large series of experiments, we concluded that there is no an one-size-fits-all solution, that can perform the best under all circumstances. In some cases a numa-oblivious implementation is more efficient while in other cases a numa-aware algorithm is needed. The configuration of execution such as the size of the structure, the number of working threads, the key range at the queue and the workload affect the performance of the examined algorithms, in not a straightforward manner, making it hard to emulate their behavior and predict which algorithm is more efficient for each case. Therefore, we utilized machine learning techniques to create a *smart* priority queue and tried to create a decision making mechanism that is capable of making these predictions.

More specifically, we modelled numa awareness as a classification problem and trained a decision tree classifier in order to solve it. We utilized our NUDDLE priority queue and its underline spraylist algorithm and defined three classes: numa-aware, numa-oblivious, and neutral (both algorithms perform similarily). We created a large dataset with many different execution configurations, trained our classifier and tuned its parameters to avoid

overfitting. Our final model can predict, with an accuracy of 93%, the most appropriate algorithm given the features of the execution (size, range, workload). Our smart priority queue uses this model in order to determine which implementation to use under specific configurations. Finally, we simulated a dynamically changing environment varying at regular intervals the benchmark's configuration and evaluated our smart priority queue. While the conditions of execution are changing, there might be a need to adapt to these changes in order to maintain the best possible performance. Our self-aware data structure can move from numa-aware to numa-oblivious algorithm and vice versa when needed, without adding further overhead and without violating correctness, as both implementations share the same core operations.

Our work can be extended in order to create a self-tunning data structure that can *on-the-fly* extract information about the features that affect its performance. In our experiments, we were aware about the changes in the configurations and the only unknown feature was the queue's size. As a future work, we could try to dynamically extract all the other features needed for the classification. A possible approach would be to dedicate a thread to periodically record information relevant to our classification problem's features such as the keys inserted to the structure, the ratio of lookup, insert and deleteMin operations, call our decision tree and notify all other threads if they need to change algorithm. Another solution would be to enrich the queue with additional fields which threads can update atomically and record statistics regarding to their operations at regular intervals.

Our work proposes a methodology to design self-aware data structures and we believe it could be a foundation for researchers to further investigate how to design dynamic concurrent data structures. Although we focused on priority queues, the methodologies we propose can be utilized for the design of other highly-contended data structures such as stacks and FIFO queues that are characterized by lower levels of parallelism and numa effect may have a high impact on their performance. Furthermore, one could experiment with random forests or other classifiers to find the most appropriate model, use techniques to boost the model's accuracy, use online learning to train the model on-the-fly or even add more features in the classification problem. Machine learning provides powerful tools that can be utilized to design highly efficient concurrent data structures, and consequently scalable concurrent algorithms.

Bibliography

[1] Yehuda Afek, Michael Hakimi, and Adam Morrison. Fast and scalable rendezvousing. In David Peleg, editor, *Distributed Computing*, pages 16–31, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[2] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. The spraylist: A scalable relaxed priority queue. *SIGPLAN Not.*, 50(8):11–20, January 2015.

[3] Gerth Stølting Brodal, Jesper Larsson Träff, and Christos D. Zaroliagis. A parallel priority queue with constant time operations. *J. Parallel Distrib. Comput.*, 49(1):4–21, February 1998.

[4] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. *SIGPLAN Not.*, 45(5):257–268, January 2010.

[5] Irina Calciu, Justin Gottschlich, and Maurice Herlihy. Using elimination and delegation to implement a scalable numa-friendly stack. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Parallelism*, San Jose, CA, 2013. USENIX.

[6] Irina Calciu, Hammurabi Mendes, and Maurice Herlihy. The adaptive priority queue with elimination and combining. In Fabian Kuhn, editor, *Distributed Computing*, pages 406–420, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[7] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-box concurrent data structures for numa architectures. *SIGPLAN Not.*, 52(4):207–221, April 2017.

[8] Tyler Crain, Vincent Gramoli, and Michel Raynal. No hot spot non-blocking skip list. In *ICDCS*, pages 196–205. IEEE Computer Society, 2013.

[9] Henry Daly, Ahmed Hassan, Michael F. Spear, and Roberto Palmieri. NUMASK: High Performance Scalable Skip List for NUMA. In Ulrich Schmid and Josef Widder, editors, *32nd International Symposium on Distributed Computing (DISC 2018)*, volume 121 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:19, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[10] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. *SIGARCH Comput. Archit. News*, 43(1):631–644, March 2015.

[11] David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: A general technique for designing numa locks. *ACM Trans. Parallel Comput.*, 1(2):13:1–13:42, February 2015.

[12] Ian Dick, Alan Fekete, and Vincent Gramoli. A skip list for multicore. *Concurrency and Computation: Practice and Experience*, 29(4), 2017.

[13] Dana Drachsler, Martin Vechev, and Eran Yahav. Practical concurrent binary search trees via logical ordering. *SIGPLAN Not.*, 49(8):343–356, February 2014.

[14] Jonathan Eastep, David Wingate, and Anant Agarwal. Smart data structures: An online machine learning approach to multicore data structures. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pages 11–20, New York, NY, USA, 2011. ACM.

[15] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 131–140, New York, NY, USA, 2010. ACM.

[16] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, 2004.

[17] Keir Fraser. Practical lock-freedom. 2004.

[18] Aurlien Gron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, Inc., 1st edition, 2017.

[19] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 355–364, New York, NY, USA, 2010. ACM.

[20] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A simple optimistic skiplist algorithm. In *Proceedings of the 14th International Conference on Structural Information and Communication Complexity*, SIROCCO'07, pages 124–138, Berlin, Heidelberg, 2007. Springer-Verlag.

[21] Baptiste Lepers. *Improving performance on NUMA systems*. Theses, Université de Grenoble, January 2014.

[22] Jonatan Lindén and Bengt Jonsson. A skiplist-based concurrent priority queue with minimal memory contention. In *Proceedings of the 17th International Conference on Principles of Distributed Systems - Volume 8304*, OPODIS 2013, 2013.

[23] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. *SIGPLAN Not.*, 49(8):317–328, February 2014.

[24] I.Lotan N.Shavit. Skiplist-based concurrent priority queues. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing*.

[25] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, November 2011.

[26] Darko Petrović, Thomas Ropars, and André Schiper. On the performance of delegation over cache-coherent shared memory. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking*, ICDCN '15, pages 17:1–17:10, New York, NY, USA, 2015. ACM.

[27] William Pugh. Concurrent maintenance of skip lists. Technical report, College Park, MD, USA, 1990.

[28] Hamza Rihani, Peter Sanders, and Roman Dementiev. Multiqueues: Simpler, faster, and better relaxed concurrent priority queues, 2014.

[29] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. Ffwd: Delegation is (much) faster than you think. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 342–358, New York, NY, USA, 2017. ACM.

[30] Konstantinos Sagonas and Kjell Winblad. The contention avoiding concurrent priority queue. In *Languages and Compilers for Parallel Computing*, 2017.

[31] Shavit, Yosef, and Maurice. Concurrent lock-free skiplist with wait-free contains operator, May 2011.

[32] Marina Sokolova and Guy Lapalme. A systematic analysis of performance measures for classification tasks. *Inf. Process. Manage.*, 45(4):427–437, July 2009.

[33] Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *Journal of Parallel and Distributed Computing*, 2005.

[34] Takayuki Usui, Reimer Behrends, Jacob Evans, and Yannis Smaragdakis. Adaptive locks: Combining transactions and locks for efficient concurrency. In *PACT*, pages 3–14. IEEE Computer Society, 2009.

[35] Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas. The lock-free k-lsm relaxed priority queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 277–278, 2015.

[36] Deli Zhang and Damian Dechev. A lock-free priority queue design based on multi-dimensional linked lists. *IEEE Trans. Parallel Distrib. Syst.*, 27(3):613–626, March 2016.

[37] Mingzhe Zhang, Francis C. M. Lau, Cho-Li Wang, Luwei Cheng, and Haibo Chen. Scalable adaptive numa-aware lock: Combining local locking and remote locking for efficient concurrency. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 50:1–50:2, New York, NY, USA, 2016. ACM.