ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Ροή Υ : Συστήματα Παράλληλης Επεξεργασίας

5η Άσκηση (Τελική Αναφορά)

Θέματα Συγχρονισμού σε Σύγχρονα Πολυπύρηνα Συστήματα

Κριθαρούλας Διονύσιος 03117875

Ομάδα:Parlab18

Εξάμηνο: 9ο

1.Σκοπός της Άσκηση

Σκοπός της συγκεκριμένης άσκησης είναι η εξοικείωση με την εκτέλεση προγραμμάτων σε σύγχρονα πολυπύρηνα συστήματα και η αξιολόγηση της επίδοσης τους. Συγκεκριμένα, θα εξετάσουμε πως κάποια χαρακτηριστικά της αρχιτεκτονικής του συστήματος επηρεάζουν την επίδοση των εφαρμογών που εκτελούνται σε αυτά και θα αξιολογήσουμε διαφόρους τρόπους υλοποίησης κλειδωμάτων για αμοιβαίο αποκλεισμό καθώς και διάφορες τακτικές συγχρονισμού για δομές δεδομένων.

2. Λογαριασμοί τράπεζας

Στο πρώτο μέρος της άσκησης μας δίνεται ένα πολυπύρηνο πρόγραμμα όπου κάθε νήμα εκτελεί ένα σύνολο πράξεων πάνω σε συγκεκριμένο στοιχείο ενός πίνακα που αντιπροσωπεύει τους λογαριασμούς των πελατών μίας τράπεζας. Το πρόγραμμα χρησιμοποιεί την βιβλιοθήκη Posix Threads (pthreads) για την δημιουργία και την διαχείριση πολλαπλών νημάτων.

2.1. Ερωτήσεις-Ζητούμενα

1. Υπάρχει ανάγκη για συγχρονισμό ανάμεσα στα νήματα της εφαρμογής;

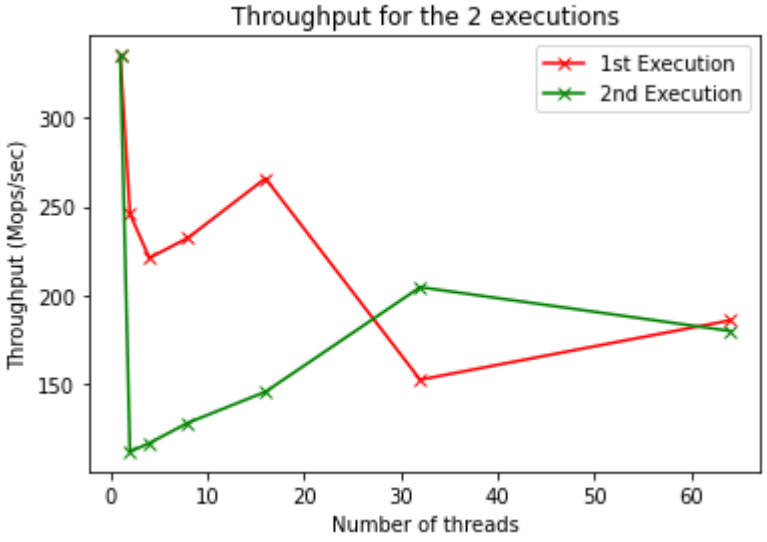
Κάθε νήμα εκτελεί το σύνολο των λειτουργιών του σε διαφορετικό λογαριασμό από όλα τα υπόλοιπα. Κανένα νήμα δεν υπάρχει περίπτωση να εκτελέσει λειτουργίες σε λογαριασμό στον οποίο εκτελεί επίσης λειτουργίες κάποιο άλλο νήμα. Επιπλέον η πρόσβαση στον κάθε λογαριασμό i γίνεται απευθείας μέσω του δείκτη του πίνακα accounts[i] με αποτέλεσμα το κάθε νήμα να έχει πρόσβαση στον λογαριασμό του απευθείας χωρίς να διαπερνά άλλες θέσεις. Έτσι δεν υπάρχει ανάγκη για συγχρονισμό ανάμεσα στα νήματα της εφαρμογής. Παρόλα αυτά η εφαρμογή μας χρησιμοποιεί barriers συγχρονίζοντας όλα τα νήματα και υποχρεώνοντας τα να περιμένουν την άδεια από το νήμα – master προκειμένου να μπορέσουν να κάνουν τα operations που επιθυμούν στον λογαριασμό τους. Ο συγχρονισμός αυτός γίνεται προκειμένου η εφαρμογή μας να μπορέσει να μετρήσει σωστά τον συνολικό χρόνο που διήρκησαν τα operations όλων των νημάτων και να βγάλει ορισμένα αποτελέσματα (Mops/sec). Ο συγχρονισμός αυτός όμως δεν είναι απαραίτητος για να λειτουργήσει ορθά η εφαρμογή μας.

2. Πώς περιμένετε να μεταβάλλεται η επίδοση της εφαρμογής καθώς αυξάνεται ο αριθμός των νημάτων;

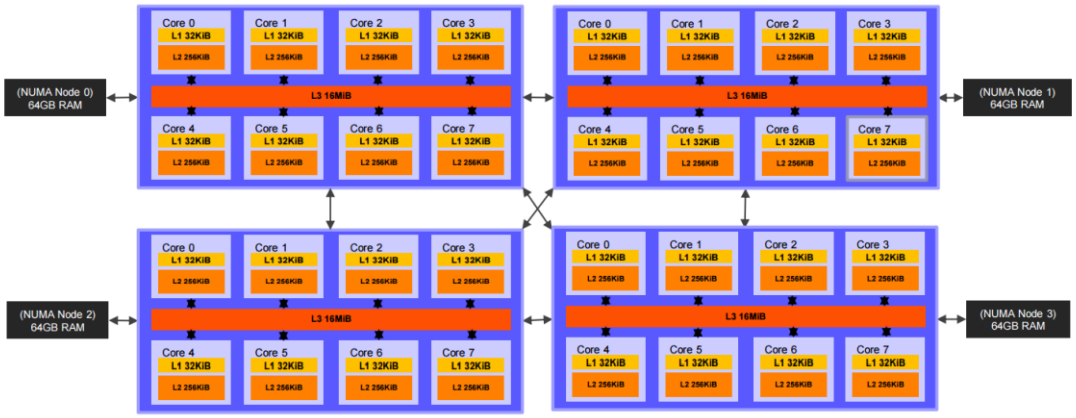
Εφόσον η εκτέλεση των λειτουργιών μεταξύ των νημάτων γίνεται ανεξάρτητα (δεν απαιτείται συγχρονισμός) περιμένουμε το throughput της εφαρμογής (Mops/sec) να αυξάνεται με την αύξηση των νημάτων.

3. Εκτελέστε την εφαρμογή με 1,2,4,8,16,32,6 νήματα χρησιμοποιώντας τις τιμές για την MT_CONF που δίνονται στον πίνακα της εκφώνησης. Δώστε ένα διάγραμμα όπου στον άξονα x θα είναι ο αριθμός των νημάτων και στον άξονα y το αντίστοιχο throughput. Το διάγραμμα θα περιέχει δύο καμπύλες, μία για κάθε εκτέλεση του πίνακα 1. Ποια είναι η συμπεριφορά της εφαρμογής για κάθε μία από τις δύο εκτελέσεις; Εξηγήστε αυτήν την συμπεριφορά και τις διαφορές ανάμεσα στις δύο εκτελέσεις.

Το διάγραμμα που προκύπτει σύμφωνα με την εκφώνηση είναι το ακόλουθο:



Σύμφωνα με το παραπάνω διάγραμμα παρατηρούμε ότι η συμπεριφορά της επίδοσης της εφαρμογής μας με την αύξηση των νημάτων δεν συμφωνεί με την αρχική μας εκτίμηση. Για να κατανοήσουμε τον λόγο μίας τέτοιας συμπεριφοράς πρέπει αρχικά να μελετήσουμε την αρχιτεκτονική του μηχανήματος sandman στο οποίο τρέχει η εφαρμογή μας. Η εικόνα της αρχιτεκτονικής του μηχανήματος μας είναι η ακόλουθη:



- socket0: 0-7, 32-39
- socket1: 8-15, 40-47
- socket2: 16-23, 48-55
- socket3: 24-31, 56-63

Παρατηρούμε ότι το μηχάνημα sandman αποτελείται από 4 sockets (CPUs) σε αρχιτεκτονική NUMA (non uniform memory access) όπου στο κάθε socket υπάρχουν 8 πυρήνες. Στον κάθε πυρήνα μπορούν να τρέχουν δύο threads (multithreading). Συνολικά λοιπόν το μηχάνημα αποτελείται από 32 πυρήνες και 64 threads.

Εστιάζοντας στην κατανομή των νημάτων έχουμε τις εξής παρατηρήσεις:

- Τα ζευγάρια threads 0-32 , 1-33 , 2-34 , … , 31-63 τρέχουν στον ίδιο πυρήνα του ίδιου socket και επομένως μοιράζονται όλα τα επίπεδα της ιεραρχίας μνήμης (L1 , L2 , L3 cache).
- Τα threads 0-7 , 32-39 τρέχουν στο socket 0. Τα threads 8-15 , 40-47 τρέχουν στο socket 1. Τα threads 16-23 , 48-55 τρέχουν στο socket 2. Τα threads 24-31 , 56-63 τρέχουν στο socket 3. Τα threads του ίδιου socket μοιράζονται την L3-cache. Threads ενός socket μπορούν να έχουν πρόσβαση σε δεδομένα που είναι αποθηκευμένα στην μνήμη RAM ενός άλλου socket (NUMA αρχιτεκτονική) με μεγαλύτερο κόστος.

Στο πρώτο σενάριο εκτέλεσης προσπαθούμε να αναθέσουμε την εκτέλεση των νημάτων σε πυρήνες που βρίσκονται στο ίδιο socket. Σε περίπτωση που τα νήματα μας δεν χωράνε σε ένα socket αναθέτουμε την εκτέλεση των επιπλέον νημάτων στους πυρήνες του επόμενου socket. Αντίθετα στο δεύτερο σενάριο εκτέλεσης τα νήματα μας αναθέτονται σε πυρήνες διαφορετικών sockets προσπαθώντας να τα κρατήσουμε όσο γίνεται πιο απομακρυσμένα μεταξύ τους.

Μετά από τις παρατηρήσεις αυτές λοιπόν μπορούμε να προχωρήσουμε στην ερμηνεία του παραπάνω διαγράμματος:

- Παρατηρούμε αρχικά ότι και στα δύο σενάρια εκτέλεσης η αύξηση των νημάτων δεν προκαλεί και ανάλογη αύξηση του throughput όπως αρχικά είχαμε υποθέσει. Αυξάνοντας τα νήματα από 1 σε 2 παρατηρούμε μεγάλη μείωση του throughput ενώ περαιτέρω αύξηση των νημάτων προκαλεί από πολύ μικρή αύξηση στο throughput μέχρι και μείωση του ($1^η$ εκτέλεση). Ο λόγος που συμβαίνει κάτι τέτοιο είναι ο τρόπος αποθήκευσης του πίνακα accounts με τους λογαριασμούς. Ο πίνακας αυτός έχει μέγεθος όσο ο αριθμός των threads (nthreads) και σε κάθε στοιχείο του αποθηκεύεται ένας μη προσημασμένος ακέραιος αριθμός των 4 byte. Έτσι το συνολικό μέγεθος του πίνακα accounts είναι 4*nthreads bytes και ως πίνακας αποθηκεύεται σε συνεχόμενες θέσεις μνήμης. Το πρόβλημα είναι ότι το block της cache του κάθε πυρήνα είναι μεγαλύτερο από τα 4bytes του κάθε στοιχείου του πίνακα accounts. Έτσι όταν ένα νήμα κάνει ένα operation (διαβάζει , γράφει) στο στοιχείο του πίνακα accounts που του αναλογεί μεταφέρει και άλλα στοιχεία του πίνακα στην cache του (ένα ολόκληρο block). Καθώς υπάρχουν ταυτόχρονα και άλλα νήματα που τρέχουν σε διαφορετικούς πυρήνες (του ίδιου αλλά και διαφορετικών sockets) απαιτείται να λειτουργήσει το πρωτόκολλο συνάφειας κρυφής μνήμης (cache coherence protocol) προκειμένου τα δεδομένα να είναι συνεπή στην cache του κάθε πυρήνα (L1 , L2) αλλά και του κάθε socket (L3). Έτσι χρειάζεται να μεταφέρονται συνεχώς δεδομένα μεταξύ των caches των πυρήνων περνώντας πιθανώς και από την κύρια μνήμη. Το ταυτόχρονο και συνεχόμενο λοιπόν γράψιμο από νήματα ανεξάρτητων θέσεων του πίνακα accounts οι οποίες όμως έχουν μεταφερθεί στις caches άλλων πυρήνων επιβάλλει το πρωτόκολλο συνάφειας μνήμης να λειτουργεί συνεχώς καθυστερώντας τον χρόνο εκτέλεσης των operations των threads. Έτσι εξηγείται η συμπεριφορά που παρατηρείται στο throughput της εφαρμογής με την αύξηση του αριθμού των νημάτων και για τις δύο εκτελέσεις.

- Παρατηρούμε ότι η $2^η$ εκτέλεση παρουσιάζει χειρότερο throughput από την πρώτη για τον ίδιο αριθμό νημάτων. Κάτι τέτοιο εξηγείτε με βάση την παραπάνω ανάλυση. Καθώς όπως είπαμε η δεύτερη εκτέλεση τοποθετεί τα νήματα σε όσο το δυνατόν πιο απομακρυσμένους sockets μεταξύ τους το πρωτόκολλο συνάφειας μνήμης απαιτείται να μεταφέρει τα δεδομένα μεταξύ διαφορετικών sockets. Η μεταφορά αυτή απαιτεί πολύ περισσότερο χρόνο (NUMA αρχιτεκτονική) συγκριτικά με την πρώτη εκτέλεση όπου τα νήματα τοποθετούνται στο ίδιο socket και μόνο όταν δεν χωράνε ανατίθενται στο επόμενο κατά σειρά.

- Πέρα από την μεγάλη μείωση που παρατηρείται στο throughput και για τις δύο εκτελέσεις καθώς πηγαίνουμε από το ένα νήμα στα δύο (γεγονός που οφείλεται ακριβώς σε αυτό που αναλύσαμε για την ανάγκη συνέπειας των δεδομένων μεταξύ των caches διαφορετικών πυρήνων) παρατηρείται επίσης μία μεγάλη μείωση στην εκτέλεση 1 καθώς μεταβαίνουμε από τα 16 στα 32 threads. Μπορούμε να παρατηρήσουμε ότι μέχρι τα 16 threads όλα τα νήματα έτρεχαν στο ίδιο socket για την εκτέλεση 1 (0-7 , 32-39). Πηγαίνοντας από τα 16 στα 32 αναγκαζόμαστε να αναθέσουμε την εκτέλεση των επόμενων 16 threads σε διαφορετικό socket. Γίνεται λοιπόν φανερή η μεγάλη διαφορά στο κόστος της διατήρησης της συνέπειας που προκαλεί η εκτέλεση threads σε διαφορετικά sockets η οποία οφείλεται στην NUMA αρχιτεκτονική. Το throughput λοιπόν μειώνεται πολύ σε μία τέτοια μετάβαση εξαιτίας της αύξησης του χρόνου που απαιτείται για την μεταφορά των δεδομένων μεταξύ των caches των πυρήνων.

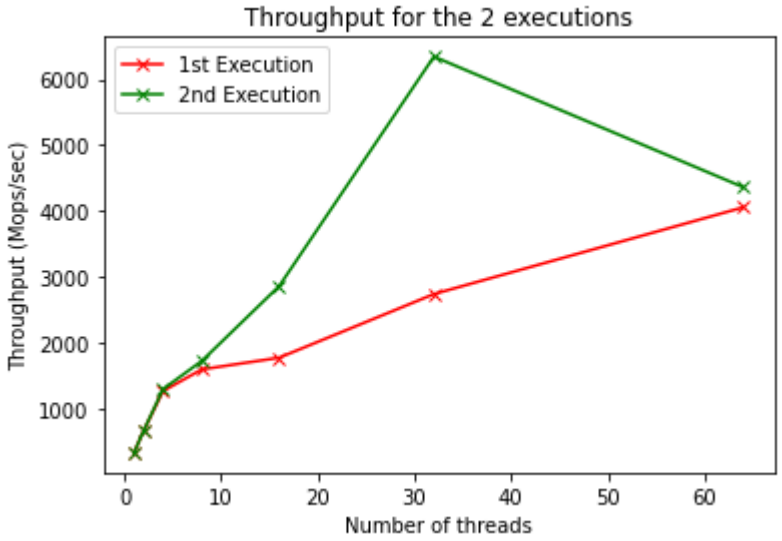
4. Η εφαρμογή έχει την συμπεριφορά που αναμένατε; Αν όχι εξηγήστε γιατί συμβαίνει αυτό και προτείνετε μία λύση. Τροποποιήστε κατάλληλα τον κώδικα και δώστε πάλι τα αντίστοιχα διαγράμματα για τις δύο εκτελέσεις.

Η εξήγηση της συμπεριφοράς της εφαρμογής αλλά και της διαφοράς μεταξύ των δύο εκτελέσεων έγινε στο προηγούμενο ερώτημα. Η λύση στο πρόβλημα που αναφέρθηκε είναι κάθε στοιχείο του πίνακα accounts να ανατίθενται σε διαφορετικό cache line έτσι ώστε όταν ένα thread γράφει ή διαβάζει το στοιχείο του πίνακα accounts που του αναλογεί να μην μεταφέρονται στην cache του πυρήνα στον οποίο τρέχει και διπλανά στοιχεία του πίνακα A. Για να πραγματοποιηθεί αυτό προσθέτουμε τον κατάλληλο padding στον ορισμό της δομής (struct) accounts του κώδικα. Το padding αυτό προσθέτει 64 – sizeof(unsigned int) bytes στο στοιχείο accounts[i] της δομής accounts καθώς 64 bytes είναι το cache block. Στα bytes αυτά δεν πρόκειται να επέμβει κάποιο thread αλλά τοποθετούνται ακριβώς για να συμπληρώσουν το μέγεθος που απαιτείται για ένα cache block. Το κομμάτι του κώδικα που εκτελεί την αλλαγή αυτή είναι το ακόλουθο:

```
/**
 * The accounts' array.
 **/
struct {
        unsigned int value;
        char padding[64-sizeof(unsigned int)];
} accounts[MAX_THREADS];
```

Σημείωση: Να σημειωθεί ότι ήδη από τον αρχικό κώδικα που μας δίνεται έτοιμος έχει προστεθεί padding στην δομή tdata_t η οποία περιέχει τα δεδομένα που παίρνει κάθε νήμα όταν καλείται από το master thread για να εκτελεστεί. Επειδή στα δεδομένα αυτά περιέχεται το πεδίο ops το οποίο αποτελεί έναν μετρητή των operations που έχει εκτελέσει το κάθε νήμα, ο μετρητής αυτός αλλάζει συνεχώς από το εκάστοτε νήμα. Μην έχοντας κάνει το padding λοιπόν θα είχαμε ακριβώς το ίδιο πρόβλημα που παρουσιάζεται και στην περίπτωση του accounts (cache coherence πρωτόκολλο και μεταφορά των δεδομένων μεταξύ των caches.).

Το διάγραμμα που προκύπτει μετά την εκτέλεση του νέου κώδικα και για τα δύο σενάρια εκτέλεσης είναι το ακόλουθο:



Παρατηρούμε ότι πλέον η εφαρμογή μας κλιμακώνει πολύ καλύτερα με την αύξηση των νημάτων όπου αυξάνεται το throughput και για τις δύο εκτελέσεις. Από την άλλη παρατηρούμε μία αλλαγή στην συμπεριφορά της εφαρμογής μας. Πλέον η δεύτερη εκτέλεση παρουσιάζει παρόμοιο throughput με την πρώτη μέχρι και για 8 νήματα ενώ για 16,32 νήματα παρουσιάζει καλύτερη συμπεριφορά σε σχέση με την πρώτη εκτέλεση. Ο λόγος που κάτι τέτοιο παρατηρείται είναι ότι πλέον έχουμε απαλλαχθεί από το overhead που θέτει το cache coherence καθώς κάθε νήμα επιδρά σε ξεχωριστώ cache block. Έτσι πλέον τα νήματα εκτελούν τα operations του εντελώς ανεξάρτητα χωρίς να υπάρχει «κρυμμένη» κάποια ανάγκη διατήρησης συνέπειας. Για 16 και 32 threads λοιπόν η πρώτη εκτέλεση προκειμένου να κρατήσει όλο και περισσότερα νήματα να τρέχουν στο ίδιο socket επιλέγει να θέσει δύο νήματα να εκτελεστούν στον ίδιο πυρήνα (multithreading). Στην περίπτωση μας όμως αυτό καθυστερεί την εκτέλεση καθώς τα νήματα αυτά εναλλάσσουν τις εντολές τους. Αντίθετα στην δεύτερη εκτέλεση κάθε νήμα είναι ενεργό σε διαφορετικό πυρήνα με αποτέλεσμα η εκτέλεση να γίνεται πιο αποδοτική. Για τον λόγο αυτό παρατηρείται και υψηλότερο throughput στα 16 και 32 νήματα. Στα 64 νήματα το throughput ξαναγίνεται το ίδιο και για τις δύο εκτελέσεις αφού πλέον τα νήματα καταλαμβάνουν

και στις δύο εκτελέσεις τους ίδιους πυρήνες. Παρατηρούμε και εδώ ότι πλέον στην δεύτερη εκτέλεση το throughput μειώνεται μεταβαίνοντας από τα 32 στα 64 νήματα εξαιτίας του multithreading που εμφανίζεται πρώτη φορά στην συγκεκριμένη εκτέλεση.

3. Αμοιβαίος αποκλεισμός – Κλειδώματα

Στο δεύτερο μέρος της άσκησης θα υλοποιήσουμε και θα αξιολογήσουμε διαφορετικούς τρόπους υλοποίησης κλειδωμάτων για αμοιβαίο αποκλεισμό. Για τους σκοπούς της άσκησης το κρίσιμο τμήμα που προστατεύεται μέσω των κλειδωμάτων που θα αξιολογήσουμε περιλαμβάνει την αναζήτηση τυχαίων στοιχείων σε μία ταξινομημένη συνδεδεμένη λίστα. Το μέγεθος της λίστας δίνεται σαν όρισμα στην εφαρμογή και καθορίζει και το μέγεθος του κρίσιμου τμήματος. Ασχολούμαστε μόνο με αναζητήσεις και όχι με άλλου είδους λειτουργίες καθώς θέλουμε το κρίσιμο τμήμα να είναι όσο το δυνατόν πιο γρήγορο εστιάζοντας αποκλειστικά στην απόδοση των κλειδωμάτων μας.

3.1.1.

Αρχικά παρουσιάζουμε τον κώδικα την κάθε υλοποίησης κλειδώματος αμοιβαίου αποκλεισμού ξεχωριστά μαζί με επεξηγήσεις. Έτσι έχουμε:

- **no_sync_lock**: Η συγκεκριμένη υλοποίηση δεν παρέχει αμοιβαίο αποκλεισμό και χρησιμοποιείται ως άνω όριο για την αξιολόγηση της επίδοσης των υπόλοιπων κλειδωμάτων.

- **pthread_lock**: Στην συγκεκριμένη υλοποίηση χρησιμοποιούμε ένα από τα κλειδώματα που παρέχεται από την βιβλιοθήκη Pthreads (pthread_spinlock_t). Πιο συγκεκριμένα με την υλοποίηση αυτή όποιο νήμα καταφέρει να μπει πρώτο στο κρίσιμο τμήμα παίρνει και το κλειδί για το τμήμα αυτό. Όσο το νήμα αυτό βρίσκεται μέσα στο κρίσιμο τμήμα της εφαρμογής μας εάν κάποιο άλλο νήμα προσπαθήσει να εισέλθει στο κρίσιμο τμήμα ουσιαστικά περιστρέφεται σε ένα loop (spinlock) ελέγχοντας σε κάθε επανάληψη εάν το lock αφέθηκε προκειμένου να το χρησιμοποιήσει. Όταν το νήμα που βρισκόταν στο κρίσιμο τμήμα φθάσει στο τέλος αυτού ελευθερώνει το κλειδί ώστε να μπορέσει κάποιο άλλο νήμα να το αποκτήσει και να μπει στο κρίσιμο τμήμα. Ο κώδικας υλοποίησης του κλειδώματος αυτού δίνεται στην συνέχεια:

```c
#include "lock.h"
#include "../common/alloc.h"
#include <pthread.h>

struct lock_struct {
        pthread_spinlock_t pthread_lock;
};

lock_t *lock_init(int nthreads)
{
        lock_t *lock;

        XMALLOC(lock, 1);
        /* other initializations here. */
        if (!pthread_spin_init(&(lock->pthread_lock) , PTHREAD_PROCESS_SHARED)){
                return lock;
        }

        return NULL;

}

void lock_free(lock_t *lock)
{
        pthread_spin_destroy(&(lock->pthread_lock));
        XFREE(lock);
}

void lock_acquire(lock_t *lock)
{
        pthread_spin_lock(&(lock->pthread_lock));
}

void lock_release(lock_t *lock)
{
        pthread_spin_unlock(&(lock->pthread_lock));
}
```

- **tas_lock (test-and-set lock):** Η συγκεκριμένη υλοποίηση μας δίνεται έτοιμη. Σύμφωνα με την υλοποίηση αυτή υπάρχει μία μεταβλητή state (μεταβλητή της δομής struct lock_struct που αναπαριστά το κλείδωμα) η οποία αρχικοποιείται στην τιμή UNLOCKED όταν καλεστεί η συνάρτηση αρχικοποίησης του συγκεκριμένου lock. Την μεταβλητή αυτή διαβάζουν και τροποποιούν όλα τα νήματα με χρήση της ατομικής εντολής __sync_test_and_set. Έτσι όταν ένα νήμα θέλει να εισέλθει στο κρίσιμο τμήμα θέτει στην μεταβλητή state την τιμή LOCKED (set) και διαβάζει την προηγούμενη τιμή της μεταβλητής αυτής. Εάν η προηγούμενη τιμή της μεταβλητής ήταν UNLOCKED τότε το νήμα μπορεί να εισέλθει στο κρίσιμο τμήμα αλλιώς “σπινάρει” σε ένα while loop επαναλαμβάνοντας σε κάθε επανάληψη την ίδια διαδικασία (test and set) μέχρι η μεταβλητή state να γίνει UNLOCKED. Το σημαντικό στην υλοποίηση αυτή είναι ότι η διαδικασία test and set γίνεται με χρήση της ατομικής εντολής που αναφέραμε προηγουμένως. Έτσι είμαστε σίγουροι ότι ένα νήμα θα διαβάσει την προηγούμενη τιμή της μεταβλητής state και θα θέσει την τιμή LOCKED σε αυτήν χωρίς να παρεμβληθεί οποιαδήποτε άλλη εντολή που να αναφέρεται στην συγκεκριμένη μεταβλητή (χρήση memory fence από τον επεξεργαστή).

Εάν δεν είχαμε κάποια τέτοια εγγύηση τότε υπήρχε η πιθανότητα δύο νήματα τα οποία ταυτόχρονα θέλουν να εισέλθουν στο κρίσιμο τμήμα να διαβάσουν και τα δύο UNLOCKED και να εισέλθουν και τα δύο στο κρίσιμο τμήμα. Έτσι η υλοποίηση του κλειδώματος αυτού θα έπαυε να προσφέρει αμοιβαίο αποκλεισμό. Όταν το νήμα που βρισκόταν στο κρίσιμο τμήμα φθάσει στο τέλος αυτού θέτει στην μεταβλητή state την τιμή UNLOCKED με χρήση της ατομικής εντολής __sync _lock_release. Έτσι μετά την αλλαγή αυτή το πρώτο νήμα που θα προλάβει να διαβάσει την τιμή UNLOCKED στην μεταβλητή state θα μπορέσει να εισέλθει στο κρίσιμο τμήμα.

- **ttas_lock (test-test-and-set lock):** Η συγκεκριμένη υλοποίηση έχει ως σκοπό την βελτίωση της προηγούμενης υλοποίησης. Το πρόβλημα της προηγούμενης υλοποίησης είναι ότι κάθε νήμα συνεχώς γράφει την τιμή LOCKED στην μεταβλητή state μέχρις ότου διαβάσει την μεταβλητή UNLOCKED και καταφέρει να μπει στο κρίσιμο τμήμα. Το συνεχές αυτό γράψιμο οδηγεί στην μεταφορά δεδομένων μεταξύ των caches των διαφορετικών πυρήνων στις οποίες τρέχουν τα νήματα τα οποία έχουν φορτώσει την δομή lock (και συνεπώς την μεταβλητή state) στην cache τους. Ο λόγος της συνεχής αυτής μεταφοράς είναι η συνάφεια της κρυφής μνήμης η οποία απαιτεί την λειτουργία κάποιου πρωτόκολλου συνάφειας κρυφής μνήμης (ακριβώς όπως και στο πρώτο μέρος της εργασίας). Έτσι έχουμε υπερβολική χρήση του διαδρόμου μνήμης και overhead όσον αφορά την απόδοση της υλοποίησης μας. Για τον λόγο αυτό στην ttas_lock υλοποίηση κάθε νήμα που θέλει να εισέλθει στο κρίσιμο τμήμα τρέχει σε ένα infinite loop. Σε κάθε επανάληψη του loop αυτού απλά διαβάζει την τιμή της μεταβλητής state. Όσο η τιμή αυτή παραμένει LOCKED (κάποιο άλλο νήμα βρίσκεται εκείνη την στιγμή στο κρίσιμο τμήμα) το νήμα δεν προχωράει σε κάποια ανανέωση της μεταβλητής γλιτώνοντας έτσι την μεταφορά δεδομένων μεταξύ caches. Μόλις διαβάσει την τιμή UNLOCKED εκτελεί την ατομική εντολή __sync_test_and_set διαβάζοντας την προηγούμενη τιμή της μεταβλητής state και θέτοντας την τιμή LOCKED (set) σε αυτήν. Εάν η προηγούμενη τιμή της μεταβλητής ήταν UNLOCKED τότε το νήμα μπορεί να εισέλθει στο κρίσιμο τμήμα αλλιώς ξανά ξεκινάει από την αρχή όλη την διαδικασία που μόλις περιγράψαμε. Ο λόγος που χρησιμοποιείται και πάλι η ατομική εντολή για test_and_set παρότι το νήμα αρχικά διαβάζει UNLOCKED είναι γιατί ενδέχεται δύο νήματα ταυτόχρονα να διαβάσουν UNLOCKED. Έτσι όπως και στην περίπτωση του tas_lock η ατομική εντολή εξασφαλίζει ότι μόνο ένα νήμα από τα δύο νήματα θα διαβάσει τελικά UNLOCKED και θα μπει στο κρίσιμο τμήμα. Όπως και στην tas_lock υλοποίηση όταν το νήμα που βρισκόταν στο κρίσιμο τμήμα φθάσει στο τέλος αυτού θέτει στην μεταβλητή state την τιμή UNLOCKED με χρήση της ατομικής εντολής __sync _lock_release. Ο κώδικας υλοποίησης του κλειδώματος αυτού δίνεται στην συνέχεια:

```c
#include "lock.h"
#include "../common/alloc.h"

typedef enum {
        UNLOCKED = 0,
        LOCKED
} lock_state_t;

struct lock_struct {
        lock_state_t state;
};


lock_t *lock_init(int nthreads)
{
        lock_t *lock;

        XMALLOC(lock, 1);
        /* other initializations here. */
        lock->state = UNLOCKED;
        return lock;
}

void lock_free(lock_t *lock)
{
        XFREE(lock);
}

void lock_acquire(lock_t *lock)
{
        lock_t *l = lock;
        for(;;){
                while(lock->state == LOCKED){}
                if (__sync_lock_test_and_set(&l->state, LOCKED) == UNLOCKED){
                        return;
                }
        }
}

void lock_release(lock_t *lock)
{
        lock_t *l = lock;
        __sync_lock_release(&l->state);
}
```

- **array_lock:** Η συγκεκριμένη υλοποίηση υλοποιεί ένα array based lock (Queue locks). Σύμφωνα με την υλοποίηση στην δομή lock είναι αποθηκευμένο το πλήθος τον νημάτων (μεταβλητή size), ένας πίνακας ετικετών label κάθε στοιχείο του οποίου αντιστοιχεί σε ένα νήμα και ένας ακέραιος αριθμός tail. Πιο συγκεκριμένα κατά την αρχικοποίηση του συγκεκριμένου κλειδώματος στην μεταβλητή size αποθηκεύεται το πλήθος των νημάτων , η μεταβλητή tail παίρνει την τιμή μηδέν και ο πίνακας label αρχικοποιείται στο μηδέν (αφού πρώτα γίνεται malloc στην μνήμη) εκτός από το label[0] που γίνεται 1. Όταν ένα νήμα θέλει να εισέλθει στο κρίσιμο τμήμα διαβάζει την τιμή tail αποθηκεύοντας την στην μεταβλητή priority και την αυξάνει κατά ένα (modulo size) σε σχέση με την προηγούμενη τιμή της. Η λειτουργία αυτή γίνεται με την ατομική εντολή __sync_fetch_and_add. Η μεταβλητή priority

δείχνει ουσιαστικά το στοιχείο του πίνακα labels στο οποίο αντιστοιχεί το συγκεκριμένο νήμα. Το νήμα λοιπόν διαβάζει την τιμή του στοιχείου labels[priority] και σε περίπτωση που αυτή είναι ένα εισέρχεται στο κρίσιμο τμήμα αλλιώς περιμένει μέχρι αυτή να γίνει 1. Εφόσον αρχικά μόνο το στοιχείο lables[0] έχει την τιμή 1 (ενώ όλα τα άλλα μηδέν) σημαίνει ότι μόνο το πρώτο νήμα μπορεί να εισέλθει στο κρίσιμο τμήμα ενώ όλα τα άλλα περιμένουν. Όταν ένα νήμα θελήσει να βγεί από το κρίσιμο τμήμα θέτει το lables[priority] σε μηδέν ενώ το labeles[(priority+1) mod size] σε 1. Έτσι επιτρέπει στο επόμενο νήμα το οποίο θέλησε πρώτο να μπει στο κρίσιμο τμήμα μετά από το νήμα που εξέρχεται να εισέλθει πλέον στο κρίσιμο τμήμα. Ο λόγος χρήσης της ατομικής εντολής __sync_fetch_and_add είναι πως δεν θέλουμε δύο νήματα ταυτόχρονα να διαβάσουν και να αυξήσουν την μεταβλητή tail κατά ένα. Τότε και τα δύο νήματα αυτά θα έχουν την ίδια τιμή στην μεταβλητή priority και έτσι θα εισέλθουν ταυτόχρονα στο κρίσιμο τμήμα όταν το lables[priority] γίνει 1. Η συγκεκριμένη υλοποίηση έχει την βελτίωση ότι σε σχέση με την προηγούμενη κάθε νήμα ,όταν είναι σε αναμονή , επιδρά στην δικιά του μεταβλητή labels[priority] ανεξάρτητα από τα υπόλοιπα ελέγχοντας συνεχώς την τιμή της και μόνο όταν ένα νήμα βγει από το κρίσιμο τμήμα επιτρέπει στο επόμενο από αυτό (labels[(priority+1) mod size]) να εισέλθει στο κρίσιμο τμήμα. Έτσι γλυτώνουμε σε μεγάλο βαθμό προβλήματα cache coherency καθώς κάθε νήμα λειτουργεί στην δικιά του ανεξάρτητη μεταβλητή την οποία απλώς διαβάζει. Βέβαια υπάρχει περίπτωση να χρειαστεί μεταφορά δεδομένων από τον δίαυλο μνήμης όταν ένα νήμα που βγαίνει από το κρίσιμο τμήμα αφήσει το επόμενο να εισέλθει καθώς γράφει TRUE στην μεταβλητή του πίνακα labels που του αντιστοιχεί. Επιπλέον πετυχαίνουμε καλύτερο critical sections utilization καθώς κάθε νήμα δεν ελέγχει συνέχεια ένα συγκεκριμένο lock για να μπει στο κρίσιμο τμήμα αλλά ελέγχει το δικό του προσωπικό label. Όταν έρθει η ώρα να εισέλθει στο κρίσιμο τμήμα θα ειδοποιηθεί από το thread που μόλις εξήλθε. Ο κώδικας υλοποίησης του κλειδώματός αυτού δίνεται στην συνέχεια:

```c
#include "lock.h"
#include "../common/alloc.h"
#include <pthread.h>

struct lock_struct {
        int* label;
        int tail;
        int size;
};

__thread int thread_number;

lock_t *lock_init(int nthreads)
{
        lock_t *lock;

        XMALLOC(lock, 1);
        /* other initializations here. */
        lock->size = nthreads;
        XMALLOC(lock->label,nthreads);
        lock->label[0] = 1;
        int i;
        for (i =1; i<nthreads; i++){
                lock->label[i] = 0;
        lock->tail = 0;
        }

        return lock;

}

void lock_free(lock_t *lock)
{
        XFREE(lock);
}

void lock_acquire(lock_t *lock)
{
        int priority = __sync_fetch_and_add(&(lock->tail),1)%lock->size;
        thread_number = priority;
        while(!lock->label[priority]){}
}

void lock_release(lock_t *lock)
{
        int priority = thread_number;
        lock->label[priority] = 0;
        lock->label[(priority+1)%lock->size] = 1;
}
```

Παρατήρηση: Η συγκεκριμένη υλοποίηση εγγυάται και first-come-fist-served fairness σε σχέση με τις υπόλοιπες υλοποιήσεις.

- **clh_lock:** Η υλοποίηση του κλειδώματος αυτού μας δίνεται έτοιμη. Η κύρια δομή της υλοποίησης αυτής είναι η clh_node_t η οποία έχει την μεταβλητή char locked που παίρνει τιμές TRUE και FALSE (υπάρχει και το απαραίτητο

padding). Στο εξής θα αναφερόμαστε στην δομή αυτή με την λέξη κόμβο. Σε κάθε νήμα αντιστοιχεί ένας τέτοιος κόμβος όπου το lock είναι TRUE εάν το νήμα ζητάει να μπεί ή βρίσκεται στο κρίσιμο τμήμα και FALSE σε διαφορετική περίπτωση. Στον κόμβο αυτό το νήμα δείχνει με τον δείκτη myNode ενώ επιπλέον έχει και έναν δείκτη myPred που δείχνει στον κόμβο του αμέσως προηγούμενου νήματος που θέλει να μπει στο κρίσιμο τμήμα. Επιπλέον υπάρχει και ένας δείκτης clh_node_t * tail ο οποίος δείχνει στον κόμβο του τελευταίου νήματος που προσπάθησε να μπει στο κρίσιμο τμήμα μέχρι εκείνη την στιγμή. Έτσι η εκτέλεση έχει ως εξής:

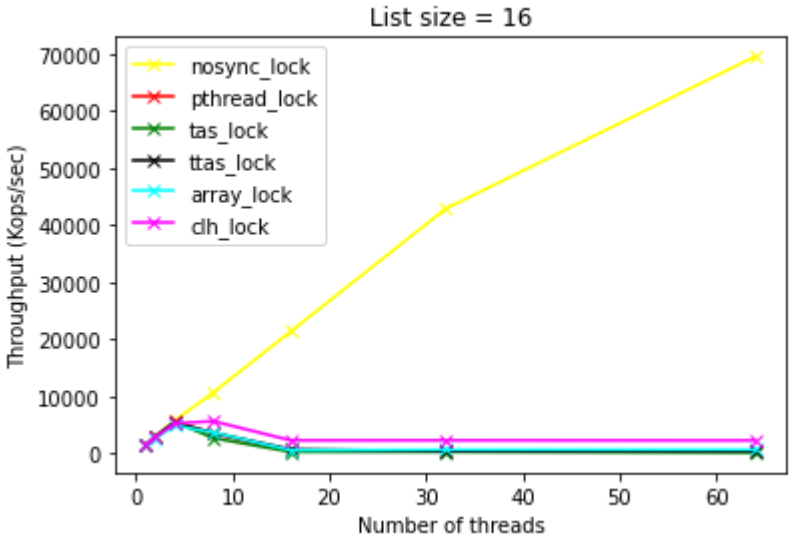
- Στην αρχικοποίηση του clh_lock γίνεται allocate το lock καθώς και ένας κόμβος με πεδίο locked = FALSE. Το tail δείχνει πλέον στον κόμβο αυτό.
- Όταν ένα νήμα θέλει να εισέλθει στο κρίσιμο τμήμα , εάν είναι η πρώτη φορά, κάνει allocate τον myNode και θέτει το myNode->lock = TRUE. Έπειτα μέσω της ατομικής εντολής __sync_test_and_set κάνει τον δείκτη myPred να δείχνει τον αμέσως προηγούμενο κόμβο και τον δείκτη tail να δείχνει τον δικό του κόμβο myNode (η χρήση της ατομικής εντολής είναι σημαντική ώστε η λειτουργία αυτή να εκτελείται ολόκληρη από ένα νήμα και μόνο).
- Εάν το myPred->lock = TRUE , που σημαίνει ότι το προηγούμενο από αυτό νήμα είτε περιμένει να μπει στο κρίσιμο τμήμα είτε βρίσκεται στο κρίσιμο τμήμα περιμένει.
- Σε διαφορετική περίπτωση εισέρχεται στο κρίσιμο τμήμα.
- Όταν ένα νήμα θελήσει να βγει από το κρίσιμο τμήμα θέτει στην μεταβλητή myNode->lock = FALSE ώστε το επόμενο κατά σειρά νήμα να μπορέσει να μπει στο κρίσιμο τμήμα και επίσης θέτει myNode = myPred ανακυκλώνοντας τον κόμβο στον οποίο αναφερόταν το myPred για μελλοντικό lock access.

Η συγκεκριμένη υλοποίηση αποτελεί βελτίωση της array_lock. Στην array_lock υλοποίηση για L locks και N threads χρειαζόμαστε 0(L*N) χώρο παρότι μπορεί το κάθε thread να κάνει access ένα lock την φορά. Ο λόγος που συμβαίνει αυτό είναι ότι με το που καλείται η αρχικοποίηση του κάθε lock γίνεται allocate ένας πίνακας με μήκος όσο το πλήθος των threads. Αντίθετα στην clh_lock υλοποίηση λόγω ακριβώς της ανακύκλωσης των κόμβων για N νήματα και L locks αν ένα νήμα κάνει access ένα lock την φορά χρειαζόμαστε 0(N+L) χώρο. Επιπλέον με την clh_lock υλοποίηση δεν είναι απαραίτητο να γνωρίζουμε από πριν τον αριθμό των threads που θα χρειαστούν το lock. Τέλος στην clh_lock υλοποίηση γλυτώνουμε και την περίπτωση της μεταφοράς δεδομένων μέσω του δίαυλου μνήμης που αναφέραμε στην array_lock υλοποίηση.

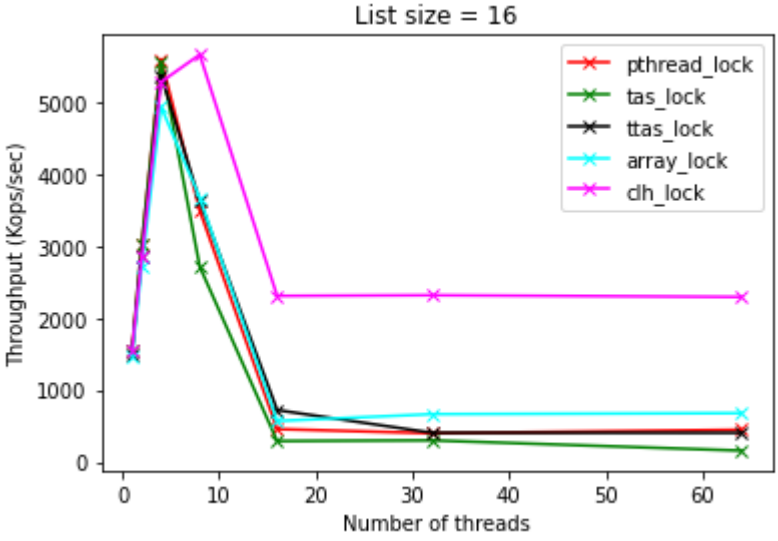
3.1.2

Στην συνέχεια εκτελούμε την εφαρμογή μας για όλα τα παραπάνω κλειδώματα ξεχωριστά. Εκτελούμε για 1,2,4,8,16,32,64 νήματα και για λίστες μεγέθους 16, 1024 , 8192. Παρουσιάζουμε τρία διαγράμματα , ένα για κάθε μέγεθος λίστας, όπου στον άξονα x είναι ο αριθμός των νημάτων και στον άξονα y το throughput της εφαρμογής:

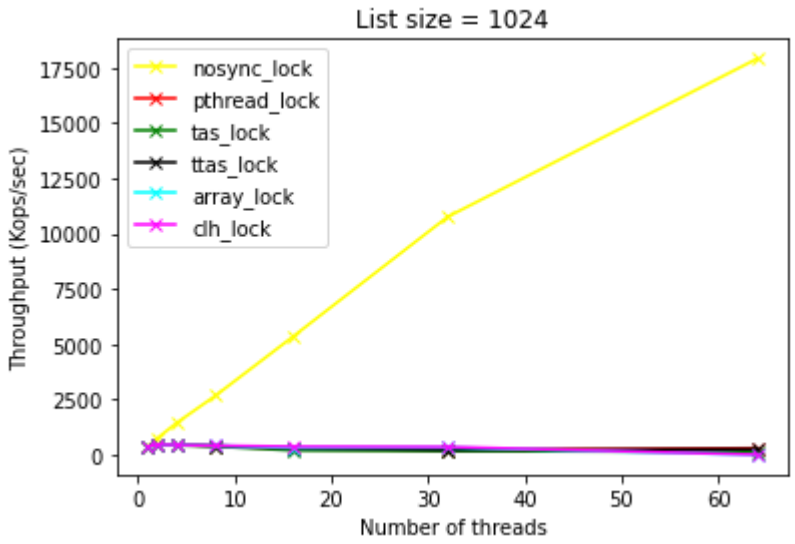
Για μήκος λίστας 16:



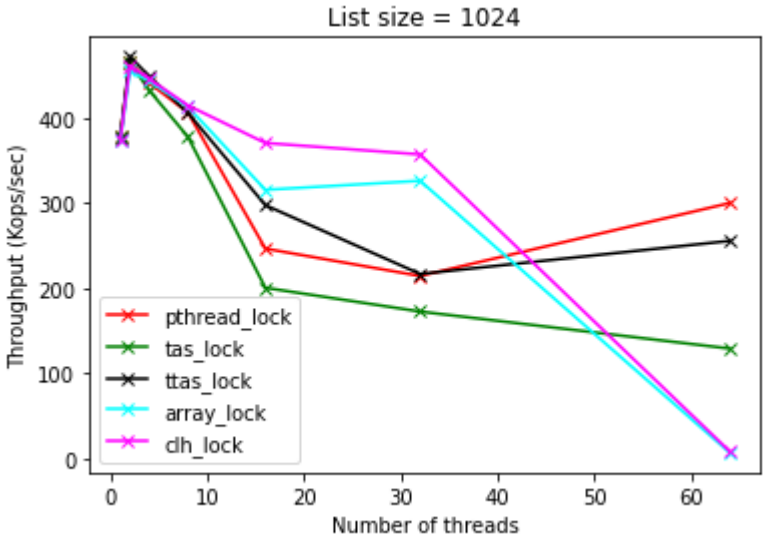
Χωρίς την nosync_list για καλύτερη εικόνα:



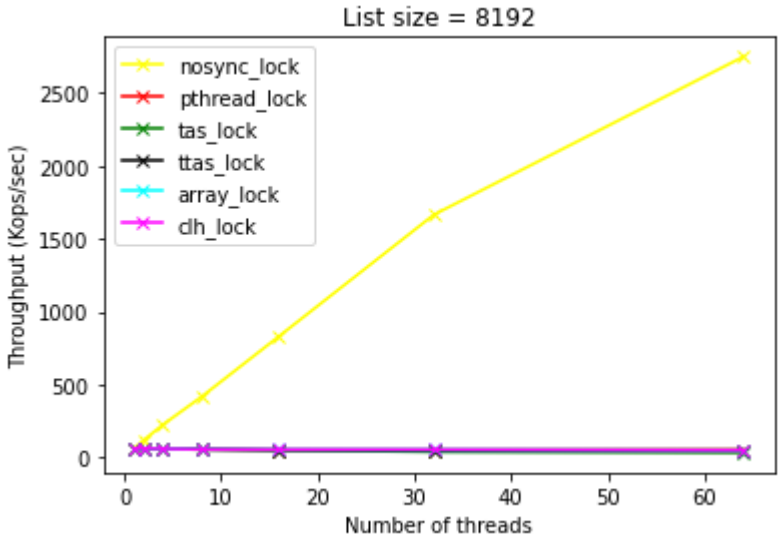
Για μήκος λίστας 1024:



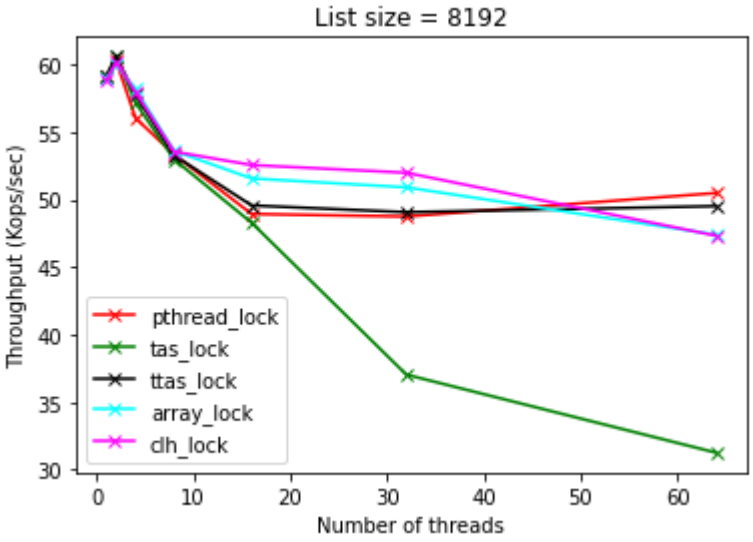
Χωρίς την nosync_list για καλύτερη εικόνα:



Για μήκος λίστας 8192:



Χωρίς την nosync_list για καλύτερη εικόνα:



Σημείωση: Σε όλες τις εκτελέσεις θέτουμε κατάλληλα την μεταβλητή περιβάλλοντος MT_ CONF ώστε τα νήματα να καταλαμβάνουν διαδοχικούς πυρήνες (π.χ. τα 16 νήματα εκτελούνται στους πυρήνες 0-15).

Παρατηρήσεις:

- Για όλες τις υλοποιήσεις κλειδωμάτων έχουμε χειρότερο throughput συγκριτικά με την nosync_lock όπου όλα τα νήματα μπαίνουν χωρίς ουσιαστικά να υπάρχει κρίσιμο τμήμα. Κάτι τέτοιο είναι προφανές καθώς η διασφάλιση του mutual exclusion προκαλεί καθυστέρηση καθώς εμποδίζει παραπάνω από ένα νήμα να βρίσκονται ταυτόχρονα στο κρίσιμο τμήμα.
- Η αύξηση του μεγέθους της λίστας μειώνει επίσης το throughput σε όλες τις υλοποιήσεις καθώς ο χρόνος που διαρκεί η εκτέλεση του κρίσιμου τμήματος από ένα νήμα είναι μεγαλύτερη μίας και το νήμα έχει να διασχίσει μεγαλύτερο κομμάτι της λίστας. Άρα έχουμε μεγαλύτερο χρόνο αναμονής για τα υπόλοιπα νήματα που περιμένουν.
- Παρατηρούμε ότι το tas lock έχει την χειρότερη επίδοση με την αύξηση των νημάτων. Κάτι τέτοιο είναι λογικό καθώς στο tas lock απαιτείται μεταφορά δεδομένων μεταξύ των caches των πυρήνων στου οποίους τρέχουν η οποία προκαλεί επιπλέον overhead όπως εξηγήσαμε και προηγουμένως. Καθώς αυξάνεται λοιπόν ο αριθμός των νημάτων αυξάνονται και οι απαιτήσεις για μεταφορά. Αντίθετα το ttas_lock το οποίο αποτελεί βελτίωση του tas_lock βλέπουμε όντως να παρουσιάζει καλύτερα αποτελέσματα.
- Το clh lock παρατηρούμε ότι για τις περισσότερες περιπτώσεις παρουσιάζει την καλύτερη επίδοση λόγο και των πλεονεκτημάτων που αναφέραμε παραπάνω.
- Το array_lock θα μπορούσαμε να πούμε ότι παρουσιάζει την δεύτερη καλύτερη επίδοση.
- Για μέγεθος λίστας 1024 και 64 νήματα παρατηρούμε όμως ότι η απόδοση των δύο παραπάνω locks μειώνεται πολύ. Αντίθετα το pthread_lock παρουσιάζει καλύτερη επίδοση με το ttas_lock να ακολουθεί. Η μεγαλύτερη πολυπλοκότητα των clh_lock και array_lock ίσως να αποτελεί αιτία της συμπεριφοράς αυτής.

4. Τακτικές Συγχρονισμού για δομές δεδομένων

Στο συγκεκριμένο μέρος της άσκησης σκοπός είναι η υλοποίηση και η αξιολόγηση των εναλλακτικών τακτικών συγχρονισμού για δομές δεδομένων. Η δομή με την οποία θα ασχοληθούμε είναι η ταξινομημένη συνδεδεμένη λίστα. Καλούμαστε να υλοποιήσουμε λίστες με τις παρακάτω 4 τακτικές συγχρονισμού:

- Fine-grain locking
- Optimistic synchronization
- Lazy synchronization
- Non-blocking synchronization

4.1. Στην συνέχεια παρατίθεται ο κώδικας της κάθε υλοποίησης:

Fine-Grain Locking:

```
#include <stdio.h>
#include <stdlib.h> /* rand() */
#include <limits.h>
#include <pthread.h> /* for pthread_spinlock_t */

#include "../common/alloc.h"
#include "ll.h"

typedef struct ll_node {
        int key;
        struct ll_node *next;
        /* other fields here? */
        pthread_spinlock_t lock;
} ll_node_t;

struct linked_list {
        ll_node_t *head;
        /* other fields here? */
};

/**
 * Create a new linked list node.
 **/
static ll_node_t *ll_node_new(int key)
{
        ll_node_t *ret;

        XMALLOC(ret, 1);
        ret->key = key;
        ret->next = NULL;
        /* Other initializations here? */
        pthread_spin_init(&(ret->lock) ,PTHREAD_PROCESS_SHARED);
        return ret;
}
```

```
/**
 * Free a linked list node.
 **/
static void ll_node_free(ll_node_t *ll_node)
{
        XFREE(ll_node);
}

/**
 * Create a new empty linked list.
 **/
ll_t *ll_new()
{
        ll_t *ret;

        XMALLOC(ret, 1);
        ret->head = ll_node_new(-1);
        ret->head->next = ll_node_new(INT_MAX);
        ret->head->next->next = NULL;

        return ret;
}

/**
 * Free a linked list and all its contained nodes.
 **/
void ll_free(ll_t *ll)
{
        ll_node_t *next, *curr = ll->head;
        while (curr) {
                next = curr->next;
                ll_node_free(curr);
                curr = next;
        }
        XFREE(ll);
}
```

```
int ll_contains(ll_t *ll, int key)
{
        int ret = 0;

        ll_node_t *pred, *curr;
        pthread_spin_lock(&(ll->head->lock));
        pred = ll->head;
        curr = pred->next;
        pthread_spin_lock(&(curr->lock));


        while (curr->key < key && curr->next != NULL){
                pthread_spin_unlock(&(pred->lock));
                pred = curr;
                curr = pred->next;
                pthread_spin_lock(&(curr->lock));
        }

        ret = (key == curr->key);

        pthread_spin_unlock(&(curr->lock));
        pthread_spin_unlock(&(pred->lock));
        return ret;
}
```

```
int ll_add(ll_t *ll, int key)
{
        int ret = 0;
        ll_node_t *pred, *curr;
        ll_node_t *new_node;

        pthread_spin_lock(&(ll->head->lock));
        pred = ll->head;
        curr = pred->next;
        pthread_spin_lock(&(curr->lock));

        while (curr->key < key && curr->next != NULL){
                pthread_spin_unlock(&(pred->lock));
                pred = curr;
                curr = pred->next;
                pthread_spin_lock(&(curr->lock));
        }

        if (key != curr->key) {
                ret = 1;
                new_node = ll_node_new(key);
                new_node->next = curr;
                pred->next = new_node;
        }

        pthread_spin_unlock(&(curr->lock));
        pthread_spin_unlock(&(pred->lock));

        return ret;
}
```

```
int ll_remove(ll_t *ll, int key)
{
        int ret = 0;


        ll_node_t *pred, *curr;

        pthread_spin_lock(&(ll->head->lock));
        pred = ll->head;
        curr = pred->next;
        pthread_spin_lock(&(curr->lock));

        while (curr->key < key && curr->next != NULL){
                pthread_spin_unlock(&(pred->lock));
                pred = curr;
                curr = pred->next;
                pthread_spin_lock(&(curr->lock));
        }

        if (key == curr->key) {
                ret = 1;
                pred->next = curr->next;
                //ll_node_free(next);
        }

        pthread_spin_unlock(&(curr->lock));
        pthread_spin_unlock(&(pred->lock));

        return ret;

}
```

```
/**
 * Print a linked list.
 **/
void ll_print(ll_t *ll)
{
        ll_node_t *curr = ll->head;
        printf("LIST [");
        while (curr) {
                if (curr->key == INT_MAX)
                        printf(" -> MAX");
                else
                        printf(" -> %d", curr->key);
                curr = curr->next;
        }
        printf(" ]\n");
}
```

Optimistic synchronization:

```
#include <stdio.h>
#include <stdlib.h> /* rand() */
#include <limits.h>
#include <pthread.h> /* for pthread_spinlock_t */

#include "../common/alloc.h"
#include "ll.h"

typedef struct ll_node {
        int key;
        struct ll_node *next;
        /* other fields here? */
        pthread_spinlock_t lock;
} ll_node_t;

struct linked_list {
        ll_node_t *head;

};

/**
 * Validate if the structure is consistent
 **/

static int ll_validate(ll_t *l , ll_node_t *pred , ll_node_t *curr)
{
        ll_node_t *node = l->head;
        while (node->key <= pred->key){
                if (node == pred){
                        return (pred->next == curr);
                }
                node = node->next;
        }
        return 0;
}
```

```
/**
 * Create a new linked list node.
 **/
static ll_node_t *ll_node_new(int key)
{
        ll_node_t *ret;

        XMALLOC(ret, 1);
        ret->key = key;
        ret->next = NULL;
        /* Other initializations here? */
        pthread_spin_init(&(ret->lock) ,PTHREAD_PROCESS_SHARED);
        return ret;
}

/**
 * Free a linked list node.
 **/
static void ll_node_free(ll_node_t *ll_node)
{
        XFREE(ll_node);
}

/**
 * Create a new empty linked list.
 **/
ll_t *ll_new()
{
        ll_t *ret;

        XMALLOC(ret, 1);
        ret->head = ll_node_new(-1);
        ret->head->next = ll_node_new(INT_MAX);
        ret->head->next->next = NULL;

        return ret;
}
```

```
/**
 * Free a linked list and all its contained nodes.
 **/
void ll_free(ll_t *ll)
{
        ll_node_t *next, *curr = ll->head;
        while (curr) {
                next = curr->next;
                ll_node_free(curr);
                curr = next;
        }
        XFREE(ll);
}

int ll_contains(ll_t *ll, int key)
{
        int ret = 0;
        ll_node_t *pred, *curr;

        while(1){

                pred = ll->head;
                curr = pred->next;

                while (curr->key < key){
                        pred = curr;
                        curr = pred->next;
                }

                if (ll_validate(ll,pred,curr)){
                        ret = (key == curr->key);
                        return ret;
                }

        }
}
```

```
int ll_add(ll_t *ll, int key)
{
        int ret = 0;
        ll_node_t *pred, *curr;
        ll_node_t *new_node;

        //printf("Entry in add method ...\n");
        while(1){

                pred = ll->head;
                curr  = pred->next;

                while (curr->key < key){
                        pred = curr;
                        curr = pred->next;
                }

                pthread_spin_lock(&(pred->lock));
                pthread_spin_lock(&(curr->lock));

                if (ll_validate(ll,pred,curr)){
                        //printf("Find valid in add...\n");
                        if (key != curr->key) {
                                ret = 1;
                                new_node = ll_node_new(key);
                                new_node->next = curr;
                                pred->next = new_node;
                        }

                        pthread_spin_unlock(&(curr->lock));
                        pthread_spin_unlock(&(pred->lock));
                        //printf("Exit add...\n");
                        return ret;
                }
                pthread_spin_unlock(&(curr->lock));
                pthread_spin_unlock(&(pred->lock));
                //printf("Dont find valid in add....\n");
        }
}
```

```
int ll_remove(ll_t *ll, int key)
{
        int ret = 0;
        ll_node_t *pred, *curr;

        //printf("Entry in remove..\n");
        while(1){
                pred = ll->head;
                curr = pred->next;
                while (curr->key < key) {
                        pred = curr;
                        curr = pred->next;
                }

                pthread_spin_lock(&(pred->lock));
                pthread_spin_lock(&(curr->lock));

                if (ll_validate(ll,pred,curr)){

                        //printf("Find calid in remove...\n");
                        if (key == curr->key) {
                                ret = 1;
                                pred->next = curr->next;
                                //ll_node_free(next);
                        }

                        pthread_spin_unlock(&(curr->lock));
                        pthread_spin_unlock(&(pred->lock));
                        //printf("Exit remove...\n");
                        return ret;
                }

                pthread_spin_unlock(&(curr->lock));
                pthread_spin_unlock(&(pred->lock));
                //printf("dont dinf valid in remove...\n");
        }

}
```

```
/**
 * Print a linked list.
 **/
void ll_print(ll_t *ll)
{
        ll_node_t *curr = ll->head;
        printf("LIST [");
        while (curr) {
                if (curr->key == INT_MAX)
                        printf(" -> MAX");
                else
                        printf(" -> %d", curr->key);
                curr = curr->next;
        }
        printf(" ]\n");
}
```

Lazy synchronization:

```
#include <stdio.h>
#include <stdlib.h> /* rand() */
#include <limits.h>
#include <pthread.h> /* for pthread_spinlock_t */

#include "../common/alloc.h"
#include "ll.h"

typedef struct ll_node {
        int key;
        struct ll_node *next;
        /* other fields here? */
        pthread_spinlock_t lock;
        int marked;
} ll_node_t;

struct linked_list {
        ll_node_t *head;
        /* other fields here? */
};


static int ll_validate(ll_node_t *pred , ll_node_t *curr)
{
       return (!pred->marked && !curr->marked && pred->next == curr);
}


/**
 * Create a new linked list node.
 **/
static ll_node_t *ll_node_new(int key)
{
        ll_node_t *ret;

        XMALLOC(ret, 1);
        ret->key = key;
        ret->next = NULL;
        /* Other initializations here? */
        pthread_spin_init(&(ret->lock), PTHREAD_PROCESS_SHARED);
        ret->marked = 0;
        return ret;
}
```

```
/**
 * Free a linked list node.
 **/
static void ll_node_free(ll_node_t *ll_node)
{
        XFREE(ll_node);
}

/**
 * Create a new empty linked list.
 **/
ll_t *ll_new()
{
        ll_t *ret;

        XMALLOC(ret, 1);
        ret->head = ll_node_new(-1);
        ret->head->next = ll_node_new(INT_MAX);
        ret->head->next->next = NULL;

        return ret;
}

/**
 * Free a linked list and all its contained nodes.
 **/
void ll_free(ll_t *ll)
{
        ll_node_t *next, *curr = ll->head;
        while (curr) {
                next = curr->next;
                ll_node_free(curr);
                curr = next;
        }
        XFREE(ll);
}
```

```
int ll_contains(ll_t *ll, int key)
{
        ll_node_t *curr = ll->head;
        int ret = 0;

        while (curr->key < key)
                curr = curr->next;

        ret = (key == curr->key && !curr->marked);
        return ret;

}
```

```
int ll_remove(ll_t *ll, int key)
{
        int ret = 0;
        ll_node_t *pred, *curr;

        //printf("Entry in remove..\n");
        while(1){
                pred = ll->head;
                curr = pred->next;
                while (curr->key < key) {
                        pred = curr;
                        curr = pred->next;
                }

                pthread_spin_lock(&(pred->lock));
                pthread_spin_lock(&(curr->lock));

                if (ll_validate(pred,curr)){

                        //printf("Find calid in remove...\n");
                        if (key == curr->key) {
                                ret = 1;
                                curr->marked = 1;
                                pred->next = curr->next;
                                //ll_node_free(next);
                        }

                        pthread_spin_unlock(&(curr->lock));
                        pthread_spin_unlock(&(pred->lock));
                        //printf("Exit remove...\n");
                        return ret;
                }

                pthread_spin_unlock(&(curr->lock));
                pthread_spin_unlock(&(pred->lock));
                //printf("dont dinf valid in remove...\n");
        }


}
```

```
int ll_add(ll_t *ll, int key)
{
        int ret = 0;
        ll_node_t *pred, *curr;
        ll_node_t *new_node;

        //printf("Entry in add method ...\n");
        while(1){

                pred = ll->head;
                curr  = pred->next;

                while (curr->key < key){
                        pred = curr;
                        curr = pred->next;
                }

                pthread_spin_lock(&(pred->lock));
                pthread_spin_lock(&(curr->lock));

                if (ll_validate(pred,curr)){
                        //printf("Find valid in add...\n");
                        if (key != curr->key) {
                                ret = 1;
                                new_node = ll_node_new(key);
                                new_node->next = curr;
                                pred->next = new_node;
                        }

                        pthread_spin_unlock(&(curr->lock));
                        pthread_spin_unlock(&(pred->lock));
                        //printf("Exit add...\n");
                        return ret;
                }
                pthread_spin_unlock(&(curr->lock));
                pthread_spin_unlock(&(pred->lock));
                //printf("Dont find valid in add....\n");
        }

}
```

```
/**
 * Print a linked list.
 **/
void ll_print(ll_t *ll)
{
        ll_node_t *curr = ll->head;
        printf("LIST [");
        while (curr) {
                if (curr->key == INT_MAX)
                        printf(" -> MAX");
                else
                        printf(" -> %d", curr->key);
                curr = curr->next;
        }
        printf(" ]\n");
}
```

Non-blocking synchronization:

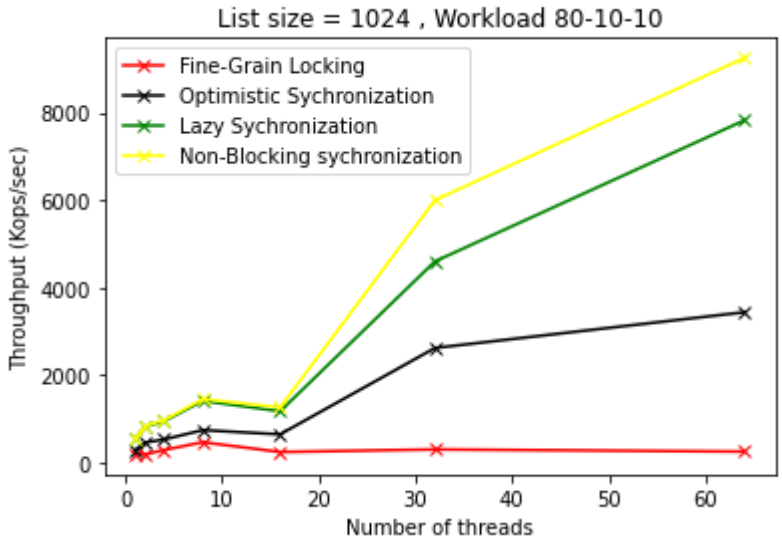
```
#include <stdio.h>
#include <stdlib.h> /* rand() */
#include <limits.h>

#include "../common/alloc.h"
#include "ll.h"

typedef struct ll_node {
        int key;
        struct ll_node *next;
        /* other fields here? */
        int marked;
} ll_node_t;

struct linked_list {
        ll_node_t *head;
        /* other fields here? */
};

/**
 * Create a new linked list node.
 **/
static ll_node_t *ll_node_new(int key)
{
        ll_node_t *ret;

        XMALLOC(ret, 1);
        ret->key = key;
        ret->next = NULL;
        /* Other initializations here? */
        ret->marked=0;
        return ret;
}

/**
 * Free a linked list node.
 **/
static void ll_node_free(ll_node_t *ll_node)
{
        XFREE(ll_node);
}
```

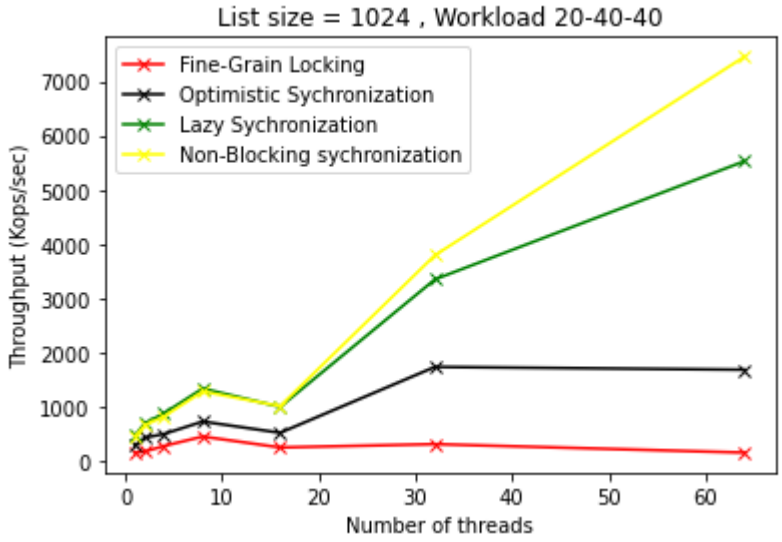
```
/**
 * Create a new empty linked list.
 **/
ll_t *ll_new()
{
        ll_t *ret;

        XMALLOC(ret, 1);
        ret->head = ll_node_new(-1);
        ret->head->next = ll_node_new(INT_MAX);
        ret->head->next->next = NULL;

        return ret;
}

/**
 * Free a linked list and all its contained nodes.
 **/
void ll_free(ll_t *ll)
{
        ll_node_t *next, *curr = ll->head;
        while (curr) {
                next = curr->next;
                ll_node_free(curr);
                curr = next;
        }
        XFREE(ll);
}
```

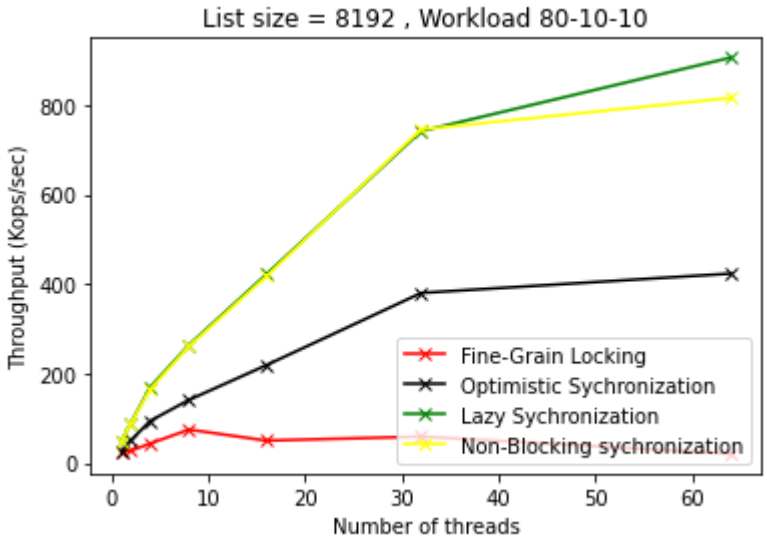
```
int ll_remove(ll_t *ll, int key)
{
        int snip;
        ll_node_t *pred;
        ll_node_t *curr;
        while(1) {
                find(ll, key, &pred, &curr);
                if (curr->key != key) {
                        return 0;
                }
                else {
                        ll_node_t *succ = curr->next;
                        snip = __sync_bool_compare_and_swap(&(curr->next), succ, curr->next);
                        if (!snip) continue;
                        curr->marked = 1;
                        __sync_bool_compare_and_swap(&(pred->next), curr, succ);
                        return 1;
                }

        }
}

int ll_add(ll_t *ll, int key)
{
        int snip;
        ll_node_t *pred;
        ll_node_t *curr;
        while(1) {
                find(ll, key, &pred, &curr);
                if (curr->key == key) {
                        return 0;
                }
                else {
                        ll_node_t * node = ll_node_new(key);
                        __sync_bool_compare_and_swap(&(node->next), node->next, curr);
                        if (__sync_bool_compare_and_swap(&(pred->next), curr, node))
                                return 1;
                }

        }

}
```

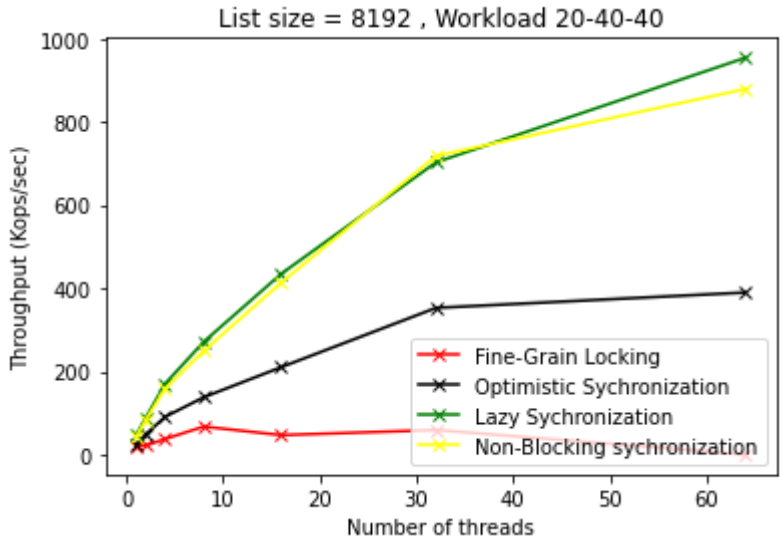
```
int ll_contains(ll_t *ll, int key)
{
        ll_node_t *curr = ll->head;
        while (curr->key < key)
                curr = curr->next;
        ll_node_t *succ = curr->next;
        return (curr->key == key) && !(curr->marked);
}

/**
 * Print a linked list.
 **/
void ll_print(ll_t *ll)
{
        ll_node_t *curr = ll->head;
        printf("LIST [");
        while (curr) {
                if (curr->key == INT_MAX)
                        printf(" -> MAX");
                else
                        printf(" -> %d", curr->key);
                curr = curr->next;
        }
        printf(" ]\n");
}
```

4.2. Στην συνέχεια παραθέτουμε τις μετρήσεις μετά την εκτέλεση της εφαρμογής για όλες τις παραπάνω υλοποιήσεις λίστας οι οποίες είναι thread safe. Πραγματοποιήθηκαν μετρήσεις της επίδοσης (Kops/sec) για 1,2,4,8,16,32,64 νήματα , για μεγέθη λίστας 1024 και 8192 και για συνδυασμούς λειτουργιών 80-10-10 και 20-40-40. Δηλαδή στην μία περίπτωση θα έχουμε 80% λειτουργίες contain και 10% λειτουργίες add και remove αντίστοιχα ενώ στην άλλη περίπτωση θα έχουμε 20% λειτουργίες contains και 40% λειτουργίες add και remove αντίστοιχα. Έτσι προκύπτουν τα ακόλουθα διαγράμματα μετρήσεων:









Παρατηρήσεις:

- Για μέγεθος λίστας 1024 και συνδυασμό λειτουργιών 80-10-10 παρατηρούμε ότι καλύτερη επίδοση έχει η non-blocking synchronization υλοποίηση η οποία δεν χρησιμοποιεί κλειδώματα. Ακολουθεί η Lazy , στην συνέχεια η optimistic και τέλος η fine-grain υλοποίηση. Μάλιστα στις δύο πρώτες υλοποιήσεις παρατηρούμε και μεγάλη κλιμάκωση καθώς αυξάνεται ο αριθμός των νημάτων ιδίως στα 32 και 64 νήματα και αύξηση της διαφοράς απόδοσης σε σχέση με τις υπόλοιπες υλοποίησης. Η fine-grain υλοποίηση παρουσιάζει την χειρότερη επίδοση καθώς η απόδοση της εφαρμογής φτάνει σε σημείο να μειώνεται με την αύξηση των νημάτων. Κάτι τέτοιο οφείλεται στα πολλά κλειδώματα που χρησιμοποιεί η συγκεκριμένη υλοποίηση και στην τεχνική hand over hand locking που χρησιμοποιεί η οποία προσδίδει μεγάλο

overhead. Η optimistic όπως ήταν αναμενόμενο παρουσιάζει καλύτερη επίδοση από την fine grain επιτρέποντας στα νήματα να διατρέχουν την λίστα χωρίς κλειδώματα και μόνο όταν βρουν τα στοιχεία που θέλουν εφαρμόζουν κλειδώματα. Έτσι βελτιώνεται το πρόβλημα του convoying που παρατηρείται στην fine-grain υλοποίηση η οποία βάζει όλα τα νήματα να ακολουθούν το πρώτο νήμα που διατρέχει την λίστα. Έτσι νήματα που θέλουν να εκτελέσουν λειτουργίες σε διαφορετικά σημεία της λίστας ίσως να μην μπορούν να τις πραγματοποιήσουν παράλληλα. Επιπλέον η lazy υλοποίηση παρουσιάζει σημαντική βελτίωση σε σχέση με την optimistic λόγω του γεγονότος ότι η συνάρτηση validate δεν διασχίζει όλη την λίστα όπως έχουμε αναλύσει και στην θεωρία αλλά χρησιμοποιείται μία επιπλέον Boolean μεταβλητή. Ακόμη στην συγκεκριμένη υλοποίηση η συνάρτηση contains δεν χρειάζεται καθόλου κλειδώματα γεγονός πάρα πολύ σημαντικό ειδικά στην περίπτωση μας όπου η κατανομή λειτουργιών είναι 80-10-10. Οι παραπάνω διαφορές και βελτιώσεις της κάθε υλοποίησης σε σχέση με την προηγούμενη γίνονται πολύ πιο φανερές με την αύξηση του αριθμού των νημάτων όπως ήταν φυσικό.

- Παρόλο που η αύξηση των νημάτων οδηγεί και σε αύξηση των συγκρούσεων οι non-blocking και lazy υλοποίηση φαίνεται να μην επηρεάζονται σημαντικά από κάτι τέτοιο. Η optimistic φαίνεται ότι επηρεάζεται μεταβαίνοντας από τα 32 στα 64 νήματα όπου η απόδοση της εφαρμογής αυξάνεται αρκετά λίγο γεγονός που οφείλεται ίσως και στην μη αποδοτική υλοποίηση της validate η οποία υποχρεώνει τα νήματα που δεν έχουν πάρει ακόμα το lock για συγκεκριμένους κόμβους να περιμένουν αρκετά μέχρι το νήμα που όχι το lock να ολοκληρώσει την κλίση του στην validate (η οποία διατρέχει όλη την λίστα) και να απελευθερώσει τα lock.
- Αλλάζοντας την κατανομή των εργασιών από 80-10-10 σε 20-40-40 και κρατώντας το μέγεθος της λίστας σταθερό και ίσο με 1024 βλέπουμε ότι και πάλι η σειρά κατάταξης των υλοποιήσεων με βάση την απόδοση τους δεν αλλάζει. Παρατηρούμε όμως μείωση της απόδοσης σε όλες τις υλοποιήσεις με μεγαλύτερη μείωση να παρατηρείται στην lazy και την optimistic υλοποίηση. Οι δύο αυτές υλοποιήσεις δεν χρησιμοποιούν κλειδώματα στην contains (ούτε η optimistic χρησιμοποιεί με αυτά που συζητήθηκαν στην θεωρία του μαθήματος) γεγονός που τις έκανε αρκετά αποδοτικές σε κατανομή λειτουργιών 80-10-10. Η αύξηση όμως των adds και των removes και η μείωση των contains οδηγεί σε μείωση της απόδοσης των συγκεκριμένων λειτουργιών ειδικά με την αύξηση των νημάτων όπου όπως αναφέραμε οι συγκρούσεις είναι περισσότερες/
- Η αύξηση του μεγέθους της λίστας από 1024 σε 8192 οδηγεί σε πολύ μεγάλη μείωση της επίδοσης και των 4 υλοποιήσεων. Όπως είναι φυσικό η αύξηση του μεγέθους οδηγεί τα νήματα να διασχίζουν μεγαλύτερα κομμάτια της λίστας γεγονός που οδηγεί στην μείωση της επίδοσης της εφαρμογής. Παρατηρούμε όμως ότι τόσο για κατανομή λειτουργιών 80-10-10 όσο και για 20-40-40 η lazy υλοποίηση παρουσιάζει την ίδια απόδοση με την non-blocking και μάλιστα ακόμα καλύτερη στα 64 νήματα. Αυτό ίσως οφείλεται στο γεγονός

πως η αύξηση του μεγέθους οδηγεί επιπλέον και σε μείωση των περιπτώσεων νήματα να εκτελούν εργασίες στα ίδια σημεία της λίστας (τυχαίες αναζητήσεις , εγγραφές και αναγνώσεις στην λίστα). Έτσι η αποτυχία της validate που θα υποχρέωνε τα νήματα να ξαναπροσπαθήσουν από την αρχή τόσο στην lazy όσο και στην optimistic υλοποίηση δεν είναι τόσο συχνή όσο στην περίπτωση του μικρότερου μεγέθους λίστας. Πέρα από τα 64 νήματα όπου παρατηρείται η διαφορά αυτή η κατάταξη των τεσσάρων αυτών υλοποιήσεων συγκριτικά με την απόδοση τους είναι η ίδια με την πρώτη περίπτωση.