ΣΥΣΤΗΜΑΤΑ ΠΑΡΑΛΛΗΛΗΣ ΕΠΕΞΕΡΓΑΣΙΑΣ

Άσκηση 3: Θέματα Συγχρονισμού σε Σύγχρονα Πολυπύρηνα Συστήματα

Χειμερινό εξάμηνο 2019-20 - Ροή Υ

Αντωνιάδης, Παναγιώτης
el15009@central.ntua.gr

Μπαζώτης, Νικόλαος
el15739@central.ntua.gr

"Redesigning your application to run multithreaded on a multicore machine is a little like learning to swim by jumping into the deep end."
– Herb Sutter, *chair of the ISO C++ standards committee*

1 Σκοπός της Άσκησης

Σκοπός της συγκεκριμένης άσκησης είναι η εξοικείωση με την εκτέλεση προγραμμάτων σε σύγχρονα πολυπύρηνα συστήματα και η αξιολόγηση της επίδοσής τους. Συγκεκριμένα, θα εξετάσουμε πώς κάποια χαρακτηριστικά της αρχιτεκτονικής του συστήματος επηρεάζουν την επίδοση των εφαρμογών που εκτελούνται σε αυτά και θα αξιολογήσουμε διάφορους 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 πολυνηματικό 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;**

   Όχι, αφού κάθε νήμα εκτελεί το σύνολο λειτουργιών του σε ξεχωριστό λογαριασμό. Αυτό πρακτικά σημαίνει ότι εκτελεί λειτουργίες σε μία συγκεκριμένη θέση του πίνακα και η πρόσβαση σε αυτήν την θέση γίνεται ανεξάρτητα από το τι συμβαίνει στον υπόλοιπο πίνακα (η πρόσβαση στην θέση i ενός πίνακα γίνεται χωρίς να διαπεράσουμε άλλες θέσεις του). Συνεπώς, κάθε νήμα μπορεί να εκτελέσει τις πράξεις που επιθυμεί ανεξάρτητα από τα άλλα χωρίς να υπάρχει ανάγκη συγχρονισμού. Να σημειωθεί ότι στο πρόγραμμα έχουν τοποθετηθεί barriers (που αποτελούν τρόπο συγχρονισμού) για να υπολογιστούν σωστά οι μετρικές που θέλουμε να καταγράψουμε και όχι γιατί υπάρχει ανάγκη συγχρονισμού στις λειτουργίες που εκτελούν τα νήματα.

2. **Πώς περιμένετε να μεταβάλλεται η επίδοση της εφαρμογής καθώς αυξάνετε τον αριθμό των νημάτων;**

   Από την στιγμή που η εκτέλεση των λειτουργιών μεταξύ των νημάτων είναι ανεξάρτητη, περιμένουμε η αύξηση του αριθμού των νημάτων να αυξήσει ανάλογα και το throughput της εφαρμογής.

3. **Εκτελέστε την εφαρμογή με 1, 2, 4, 8, 16, 32, 64 νήματα χρησιμοποιώντας τις τιμές για την MT_CONF που δίνονται στον παρακάτω πίνακα. 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.**

Αριθμός νημάτων	MT_CONF	
	Εκτέλεση 1	Εκτέλεση 2
1	0	0
2	0, 1	0, 8
4	0-3	0,8,16,24
8	0-7	0-1,8-9,16-17,24-25
16	0-7, 32-39	0-3,8-11,16-19,24-27
32	0-15, 32-47	0-31
64	0-63	0-63

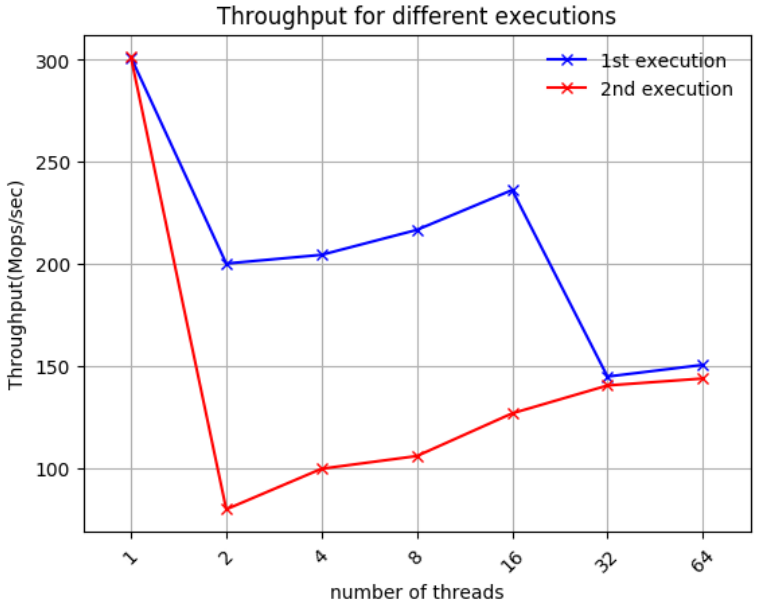
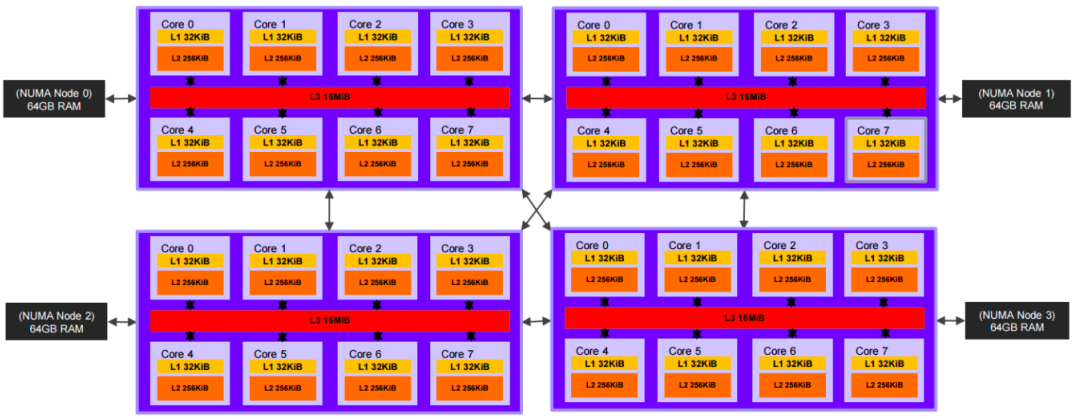
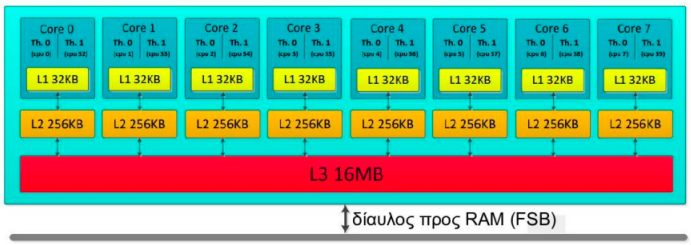


Figure 1: Throughput στις 2 εκτελέσεις

Παρατηρούμε ότι, σε αντίθεση με αυτό που περιμέναμε, η επίδοση της εφαρμογής μειώνεται καθώς προσθέτουμε νέους πυρήνες. Για να ερμηνεύσουμε τώρα την συμπεριφορά των δύο καμπυλών, πρέπει πρώτα να μελετήσουμε την αρχιτεκτονική του μηχανήματος sandman στο οποίο εκτελείται.



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



Από τις παραπάνω εικόνες, αυτό που πρέπει να παρατηρήσουμε είναι ότι:

- Τα ζευγάρια 0-32, 1-33, ..., 31-63 τρέχουν στον ίδιο πυρήνα και συνεπώς μοίραζονται όλα τα στάδια της ιεραρχίας μνήμης (Hyperthreading).
- Τα νήματα 0-7, 32-39 βρίσκονται στον ίδιο κόμβο (node 0) και μοιράζονται την L3 cache (αντίστοιχα και για τα υπόλοιπα nodes).

Παρατηρούμε ότι στην 1η εκτέλεση, χρησιμοποιούνται πυρήνες όσο πιο κοντά γίνεται ώστε να μοιράζονται κάποια κομμάτια της ιεραρχίας μνήμης και η μεταφορά ενός block από την cache του ενός στην cache του άλλου να είναι η ελάχιστη. Από την άλλη πλευρά, στην 2η εκτέλεση, ο χρόνος αυτός μεγιστοποιείται εκτελώντας τις λειτουργίες σε όσο γίνεται πιο μακρινούς πυρήνες.

Το ερώτημα τώρα είναι γιατί έχουμε συνεχή μεταφορά δεδομένων αφού οι λειτουργίες των νημάτων είναι ξεχωριστές. Στο πρόγραμμα μας, έχουμε τον πίνακα accounts ο οποίος έχει τόσες θέσεις όσα και τα νήματα και κάθε θέση του είναι ενας μη προσημασμένος ακέραιος, δηλαδή 4 bytes. Από την στιγμή που οι πίνακες αποθηκεύονται σε συνεχόμενες θέσεις μνήμης, ο πίνακας αυτός έχει μέγεθος threads*4 συνεχόμενες θέσεις. Ο λόγος που το πρόγραμμά μας δεν κλιμακώνει είναι ότι το μέγεθος

block της cache είναι μεγαλύτερο από το μέγεθος μιας θέσης του πίνακα. Συνεπώς, όταν ένα νήμα διαβάζει και γράφει σε μία θέση, φέρνει αυτό το block στην cache του και από την στιγμή που αυτό το block το είχε γράψει και κάποιο άλλο νήμα, το τρέχον νήμα πρέπει να ενημερώσει την cache του, σύμφωνα με τα γνωστό πρωτόκολλα (MESI). Συνεπώς, παρά το γεγονός ότι τα νήματα διαβάζουν και γράφουν σε διαφορετικές θέσεις μνήμης, καθυστερούν το ένα την εκτέλεση του άλλου επειδή αυτές οι θέσεις βρίσκονται στο ίδιο block της cache. Έτσι, έχουμε μία συνεχή μεταφορά δεδομένων μεταξύ των cache των πυρήνων, με αποτέλεσμα στην 2η εκτέλεση όπου οι πυρήνες είναι στις χειρότερες δυνατές θέσεις, να προκαλείται η μεγαλύτερη δυνατή καθυστέρηση για την ενημέρωση των επιπέδων της ιεραρχίας της μνήμης.

4. **Η εφαρμογή έχει την συμπεριφορά που αναμένατε; Αν όχι, εξηγήστε γιατί συμβαίνει αυτό και προτείνετε μία λύση. Τροποποιήστε κατάλληλα τον κώδικα και δώστε και πάλι τα αντίστοιχα διαγράμματα για τις δύο εκτελέσεις.**
Υπόδειξη: πώς αποθηκεύεται ο πίνακας με τους λογαριασμούς στα διάφορα επίπεδα της ιεραρχίας της μνήμης και τι προκαλεί αυτό ανάμεσα στα νήματα της εφαρμογής;

   Όπως αναφέρθηκε στο προηγούμενο ερώτημα, η καθυστέρηση προκαλείται επειδή τα νήματα διαβάζουν και γράφουν σε συνεχόμενες θέσεις μνήμης. Μία λύση στο πρόβλημα αυτό είναι να αλλάξουμε τον κώδικα, ώστε τα νήματα να γράφουν σε διαφορετικά blocks της cache. Αυτό μπορεί να συμβεί προσθέτοντας περιττά bytes ως padding σε κάθε θέση του πίνακα. Ακολουθεί η βελτιστοποιημένη δομή και το νέο διάγραμμα:

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

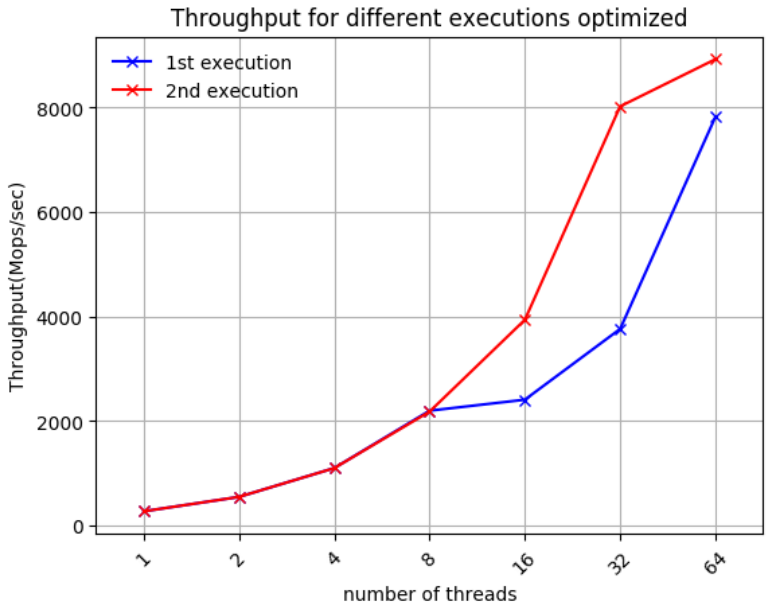


Figure 2: Throughput στις 2 εκτελέσεις

Παρατηρούμε το πρόγραμμα τώρα κλιμακώνει πολύ καλά. Επίσης, παρατηρούμε ότι η 2η εκτέλεση είναι αποδοτικότερη από την 1η για 16 και 32 νήματα. Αυτό συμβαίνει διότι στην 1η περίπτωση έχουμε 16 (και 32) νήματα να τρέχουν σε 8 (και 16) πυρήνες, έχουμε, δηλαδή, συνεχώς δύο νήματα ενεργά ανά πυρήνα. Αντίθετα, στην 2η περίπτωση για 16 (και 32) νήματα χρησιμοποιούμε 16 (και 32) πυρήνες, ένα για κάθε νήμα, με αποτέλεσμα να μην παρεμβάλλονται εντολές από 2 νήματα σε ένα πυρήνα. Ουσιαστικά, εδώ όπου όλες οι λειτουργίες είναι ανεξάρτητες και θα μπορούσαν να εκτελεστούν εντελώς παράλληλα, το hyperthreading προκαλεί καθυστερήσεις.

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 αξιολογήσουυμε 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.

3.1 Ερωτήσεις - Ζητούμενα

1. **Υλοποιήστε τα ζητούμενα κλειδώματα συμπληρώνοντας τα αντίστοιχα αρχεία της μορφής <lock_type>_lock.c.**

   Έχουμε στη διάθεσή μας τα εξής κλειδώματα:

   - **nosync_lock:** Η συγκεκριμένη υλοποίηση δεν παρέχει αμοιβαίο αποκλεισμό και χρησιμοποιείται ώς άνω όριο για την αξιολόγηση της επίδοσης των υπόλοιπων κλειδωμάτων. Η υλοποίησή του μας δίνεται.
   - **pthread_lock:** Η συγκεκριμένη υλοποίηση χρησιμοποιεί ένα από τα κλειδώματα που παρέχεται από την βιβλιοθήκη Pthreads (pthread_spinlock_t).

```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. */

        /* initialize the pthread spin lock */
        pthread_spin_init(&(lock->pthread_lock), PTHREAD_PROCESS_SHARED);

        return lock;
}

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));
}
```

```
32
33 void lock_release(lock_t *lock)
34 {
35         pthread_spin_unlock(&(lock->pthread_lock));
36 }
```

- **tas_lock:** Το test-and-set κλείδωμα όπως έχει παρουσιαστεί στις διαλέξεις του μαθήματος.
- **ttas_lock:** Το test-and-test-and-set κλείδωμα όπως έχει παρουσιαστεί στις διαλέξεις του μαθήματος. Η υλοποίησή του μας δίνεται.

```
1  #include "../common/alloc.h"
2  #include "lock.h"
3
4  typedef enum {
5          UNLOCKED = 0,
6          LOCKED
7  } lock_state_t;
8
9  struct lock_struct {
10         lock_state_t state;
11 };
12
13 lock_t *lock_init(int nthreads)
14 {
15         lock_t *lock;
16
17         XMALLOC(lock, 1);
18         lock->state = UNLOCKED;
19         return lock;
20 }
21
22 void lock_free(lock_t *lock)
23 {
24         XFREE(lock);
25 }
26
27 void lock_acquire(lock_t *lock)
28 {
29         lock_t *l = lock;
30
31         while(1) {
32                 while(l->state == LOCKED) {
33                         /* do nothing */
34                 }
35                 if (__sync_lock_test_and_set(&l->state, LOCKED) == UNLOCKED)
36                         return;
37         }
38 }
39
40 void lock_release(lock_t *lock)
41 {
42         lock_t *l = lock;
43
44         __sync_lock_release(&l->state);
45 }
```

- **array_lock:** Το array-based κλείδωμα όπως περιγράφεται και στις διαφάνειες του μαθήματος.

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

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

__thread int mySlotIndex;

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

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

        return lock;
}

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

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

void lock_release(lock_t *lock)
{
        int slot = mySlotIndex;
        lock->flag[slot] = 0;
        lock->flag[(slot+1)%lock->size] = 1;
}
```

- **clh_lock:** Ένα είδος κλειδώματος που στηρίζεται στη χρήση μίας συνδεδεμένης λίστας. Αναλυτικές πληροφορίες μπορούμε να βρούμε στο Κεφάλαιο 7 του βιβλίου "The Art of Multiprocessor Programming". Η υλοποίησή του μας δίνεται.

2. **Εκτελέστε την εφαρμογή με όλα τα διαφορετικά κλειδώματα που σας παρέχονται και αυτά που υλοποιήσατε. Εκτελέστε για 1, 2, 4, 8, 16, 32, 64 νήματα και**

για λίστες μεγέθους 16, 1024, 8192. Παρουσιάστε τα αποτελέσματά σας σε διαγράμματα αντίστοιχα με το πρώτο μέρος της άσκησης και εξηγήστε την συμπεριφορά της εφαρμογής για κάθε κλείδωμα.
Σημείωση: σε όλες τις εκτελέσεις θα θέσετε κατάλληλα την μεταβλητή περιβάλλοντος MT_CONF ώστε τα νήματα να καταλαμβάνουν διαδοχικούς πυρήνες, π.χ. τα 16 νήματα εκτελούνται στους πυρήνες 0-15.

- Μέγεθος λίστας: 16

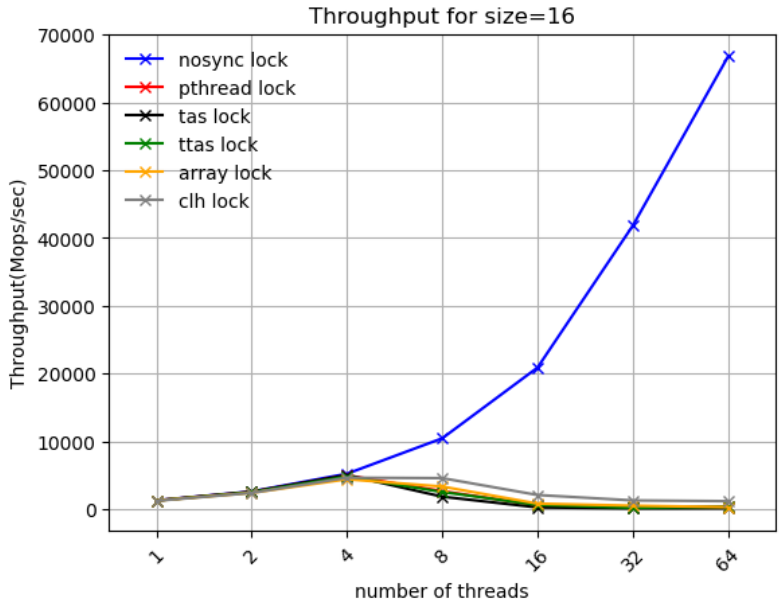


Figure 3: Throughput για διάφορα κλειδώματα και μέγεθος λίστας 16

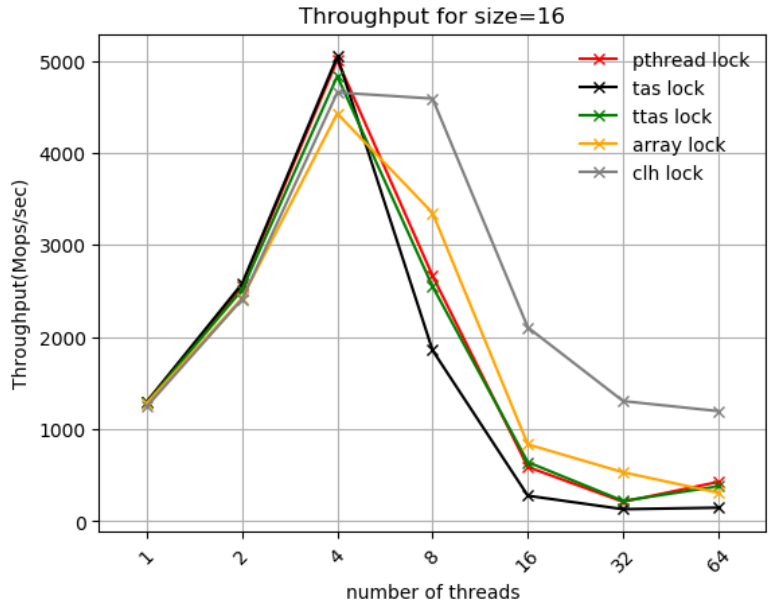


Figure 4: Αντίστοιχα με παραπάνω αλλά χωρίς το nosync lock για καλύτερη σύγκριση

- Μέγεθος λίστας: 1024

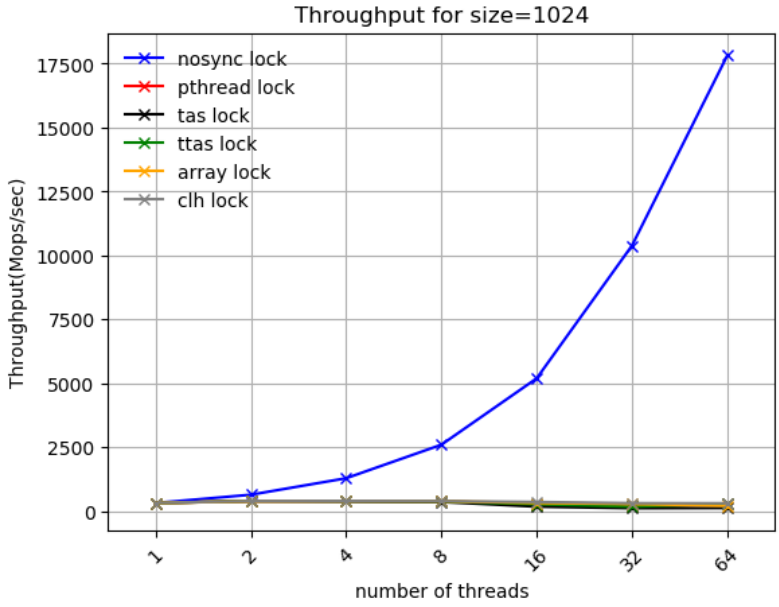


Figure 5: Throughput για διάφορα κλειδώματα και μέγεθος λίστας 1024

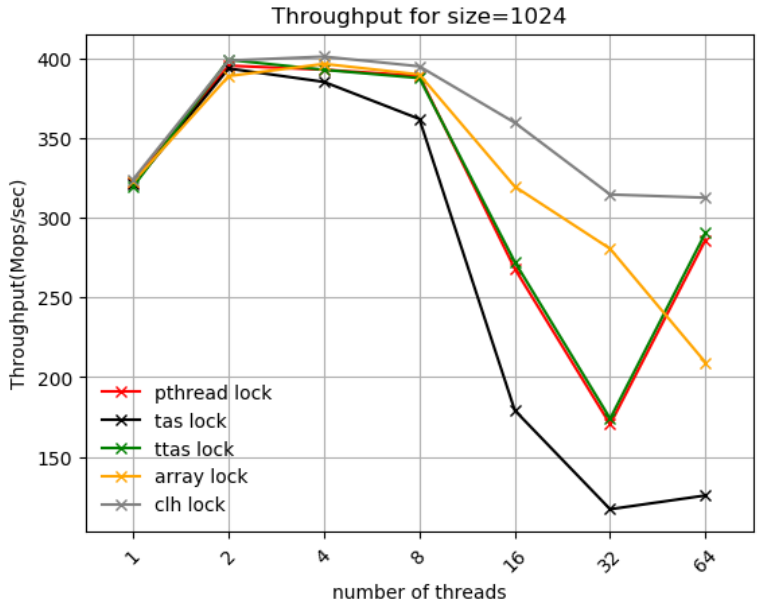


Figure 6: Αντίστοιχα με παραπάνω αλλά χωρίς το nosync lock για καλύτερη σύγκριση

- Μέγεθος λίστας: 8192

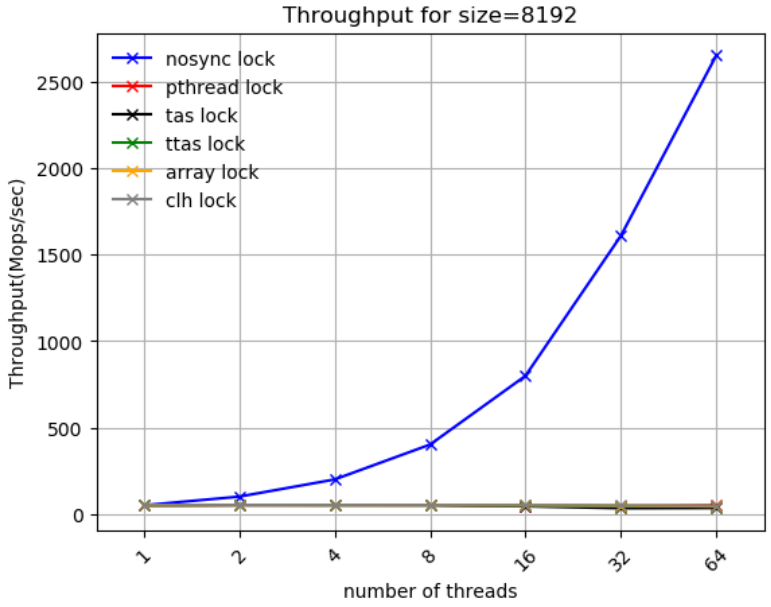


Figure 7: Throughput για διάφορα κλειδώματα και μέγεθος λίστας 8192

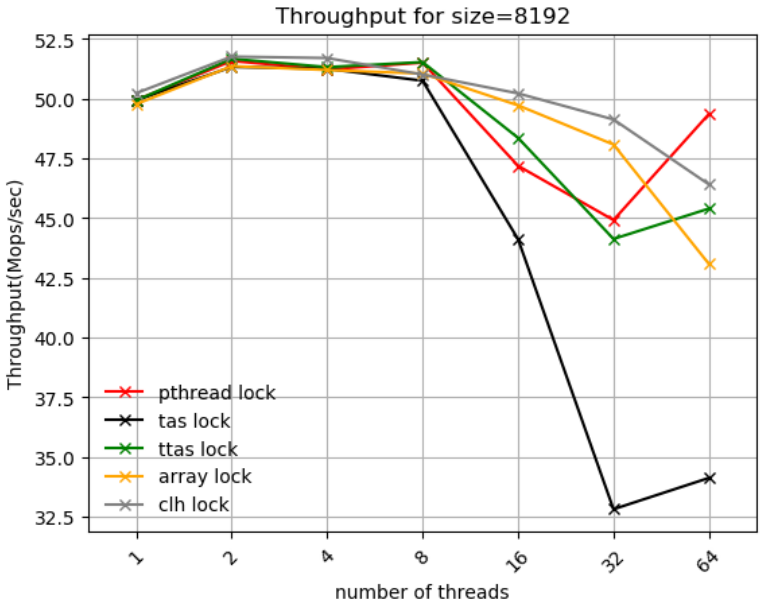


Figure 8: Αντίστοιχα με παραπάνω αλλά χωρίς το nosync lock για καλύτερη σύγκριση

Σχολιασμός:

(a) Όπως ήταν αναμενόμενο, η απόκλιση των locks από την υλοποίηση χωρίς συγχρονισμό είναι αρχικά μικρή και αυξάνεται με πολύ μεγάλο ρυθμό καθώς αυξάνονται τα νήματα. Αυτό συμβαίνει, γιατί όσα περισσότερα νήματα προσπαθούν να μπουν στο κρίσιμο τμήμα, τόσο μεγαλύτερη καθυστέρηση έχουμε στα locks καθώς τα αναγκάζουμε να εκτελέσουν το κρίσιμο τμήμα ένα ένα (σε αντίθεση με το no_sync που απλά μπαίνουν όλα μαζί στο κρίσιμο τμήμα).

(b) Επίσης, όσο μεγαλύτερη είναι η λίστα τόσο περισσότερο χρόνο παίρνει το κρίσιμο τμήμα, με αποτέλεσμα να μειώνεται η συνολική επίδοση.

(c) Σε όλες τις περιπτώσεις, σε μεγάλο αριθμό νημάτων το clh lock έχει την καλύτερη επίδοση με το array lock να ακολουθεί. Αυτό συμβαίνει, διότι στα άλλα κλειδώματα δημιουργείται μεγάλη κυκλοφορία δεδομένων από το σύστημα συνάφειας κρυφής μνήμης. Εξαίρεση αποτελεί το pthread lock για μέγεθος λίστας 8192 και 64 νήματα, το οποίο έχει την καλύτερη επίδοση.

(d) Το tas lock όσο μεγαλώνει η λίστα (και συνεπώς το κρίσιμο τμήμα) και αριθμός των νημάτων πάει όλο και χειρότερα καθώς κάνει υπερβολική χρήση του διαδρόμη για μεταφορά δεδομένων (συνάφεια κρυφής μνήμης).

(e) Το ttas lock είναι πάντα αποδοτικότερο από το tas lock επιβεβαιώνοντας την θεωρία.

Συνολικά, το συμπέρασμα είναι ότι για μικρό αριθμό νημάτων μπορούμε να χρησιμοποιήσουμε απλά κλειδώματα (pthread, tas, ttas) καθώς δεν είναι πολλοί αυτοί που διεκδιδούν μία θέση στο κρίσιμο τμήμα με αποτέλεσμα τα αρνητικά τους να μην είναι ακόμα εμφανή. Όταν αυξάνουμε όμως τα νήματα, πρέπει να χρησιμοποιήσουμε κάποιο πιο εξελιγμένο κλείδωμα (array, clh) τα

οποία χρησιμοποιώντας δομές (πίνακα, λίστα αντίστοιχα) επιτρέπουν στα νήματα να εργάζονται όσο γίνεται το καθένα στο δικό του χώρο και να μην 'πειράζουν' συνεχώς κοινές θέσεις μνήμης.

4 Τακτικές συγχρονισμού για δομές δεδομένων

Στο τρίτο και τελευταίο μέρος της άσκησης στόχος είναι η υλοποίηση και η αξιολόγηση των διαφορετικών εναλλακτικών τακτικών συγχρονισμού για δομές δεδομένων. Η δομή με την οποία ασχοληθήκαμε είναι η ταξινομημένη συνδεδεμένη λίστα.

4.1 Ερωτήσεις - Ζητούμενα

1. **Υλοποιήστε τις ζητούμενες λίστες συμπληρώνοντας τα αντίστοιχα αρχεία της μορφής ll_<sync_ type>.c.**

   - **Fine-grain locking**

```c
#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;
    pthread_spinlock_t lock;
} ll_node_t;

struct linked_list {
    ll_node_t *head;
};

/**
 * 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;
    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)
```

```
{
    pthread_spin_destroy(&(ll_node->lock));
    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;

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

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

        ret = (curr->key == key);
    pthread_spin_unlock(&(pred->lock));
    pthread_spin_unlock(&(curr->lock));
        return ret;

}
```

```
96
97  int ll_add(ll_t *ll, int key)
98  {
99      int ret = 0;
100     ll_node_t *pred, *curr;
101         ll_node_t *new_node;
102
103         pred = ll->head;
104     pthread_spin_lock(&(pred->lock));
105         curr = pred->next;
106     pthread_spin_lock(&(curr->lock));
107
108         while (curr->key < key && curr->next != NULL) {
109         pthread_spin_unlock(&(pred->lock));
110                 pred = curr;
111                 curr = curr->next;
112         pthread_spin_lock(&(curr->lock));
113         }
114
115         if (curr->key != key) {
116                 new_node = ll_node_new(key);
117                 new_node->next = curr;
118         pred->next = new_node;
119         ret = 1;
120         }
121     pthread_spin_unlock(&(curr->lock));
122     pthread_spin_unlock(&(pred->lock));
123
124         return ret;
125 }
126
127 int ll_remove(ll_t *ll, int key)
128 {
129         int ret = 0;
130         ll_node_t *pred, *curr;
131
132         pred = ll->head;
133         pthread_spin_lock(&(pred->lock));
134     curr = pred->next;
135     pthread_spin_lock(&(curr->lock));
136
137         while (curr->key < key && curr->next != NULL) {
138         pthread_spin_unlock(&(pred->lock));
139                 pred = curr;
140                 curr = curr->next;
141         pthread_spin_lock(&(curr->lock));
142         }
143
144         if (curr->key == key) {
145         pred->next = curr->next;
146         pthread_spin_unlock(&(curr->lock));
147         pthread_spin_unlock(&(pred->lock));
148                 ll_node_free(curr);
149             ret = 1;
150     }
151     else {
152         pthread_spin_unlock(&(curr->lock));
153         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;
    pthread_spinlock_t lock;
} ll_node_t;

struct linked_list {
    ll_node_t *head;
};

/**
 * 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;
    pthread_spin_init(&(ret->lock), PTHREAD_PROCESS_SHARED);

    return ret;
}
```

```
33
34 /**
35  * Free a linked list node.
36  **/
37 static void ll_node_free(ll_node_t *ll_node)
38 {
39     pthread_spin_destroy(&(ll_node->lock));
40     XFREE(ll_node);
41 }
42
43 /**
44  * Create a new empty linked list.
45  **/
46 ll_t *ll_new()
47 {
48     ll_t *ret;
49
50     XMALLOC(ret, 1);
51     ret->head = ll_node_new(-1);
52     ret->head->next = ll_node_new(INT_MAX);
53     ret->head->next->next = NULL;
54
55     return ret;
56 }
57
58 /**
59  * Free a linked list and all its contained nodes.
60  **/
61 void ll_free(ll_t *ll)
62 {
63     ll_node_t *next, *curr = ll->head;
64     while (curr) {
65         next = curr->next;
66         ll_node_free(curr);
67         curr = next;
68     }
69     XFREE(ll);
70 }
71
72
73 int ll_validate(ll_t *ll, ll_node_t *pred, ll_node_t *curr)
74 {
75     ll_node_t *node = ll->head;
76     while (node != NULL && node->key <= pred->key){
77         if (node == pred)
78             return (pred->next == curr);
79         node = node->next;
80     }
81
82     return 0;
83 }
84
85
86 int ll_contains(ll_t *ll, int key)
87 {
88     int ret = 0;
89     ll_node_t *pred, *curr;
90         while(1) {
```

```
91                  pred = ll->head;
92                  curr = pred->next;
93
94                  while (curr != NULL && curr->key <= key) {
95              if (curr->key == key)
96                  break;
97                          pred = curr;
98                          curr = curr->next;
99                  }
100                 pthread_spin_lock(&(pred->lock));
101                 pthread_spin_lock(&(curr->lock));
102
103                 if (ll_validate(ll, pred, curr)){
104                     ret = (curr->key == key);
105             pthread_spin_unlock(&(curr->lock));
106                     pthread_spin_unlock(&(pred->lock));
107             return ret;
108                 }
109         pthread_spin_unlock(&(curr->lock));
110                 pthread_spin_unlock(&(pred->lock));
111         }
112
113 }
114
115 int ll_add(ll_t *ll, int key)
116 {
117         int ret = 0;
118     ll_node_t *pred, *curr, *new_node;
119
120     while(1) {
121             pred = ll->head;
122             curr = pred->next;
123
124             while (curr != NULL && curr->key <= key) {
125             if (curr->key == key)
126                 break;
127                     pred = curr;
128                     curr = curr->next;
129             }
130         pthread_spin_lock(&(pred->lock));
131         pthread_spin_lock(&(curr->lock));
132
133         if (ll_validate(ll, pred, curr)){
134             if (curr->key == key) {
135                 pthread_spin_unlock(&(curr->lock));
136                             pthread_spin_unlock(&(pred->lock));
137                 return ret;
138             }
139                 else {
140                         ret = 1;
141                         new_node = ll_node_new(key);
142                         new_node->next = curr;
143                         pred->next = new_node;
144                 pthread_spin_unlock(&(curr->lock));
145                             pthread_spin_unlock(&(pred->lock));
146                 return ret;
147                 }
148         }
```

```
149         pthread_spin_unlock(&(curr->lock));
150                 pthread_spin_unlock(&(pred->lock));
151     }
152 
153 }
154 
155 int ll_remove(ll_t *ll, int key)
156 {
157         int ret = 0;
158     ll_node_t *pred, *curr;
159 
160         while(1) {
161 
162                 pred = ll->head;
163                 curr = pred->next;
164 
165                 while (curr != NULL && curr->key <= key) {
166             if (curr->key == key)
167                 break;
168                         pred = curr;
169                         curr = curr->next;
170                 }
171                 pthread_spin_lock(&(pred->lock));
172                 pthread_spin_lock(&(curr->lock));
173 
174                 if (ll_validate(ll, pred, curr)){
175                         if (curr->key == key) {
176                 ret = 1;
177                 pred->next = curr->next;
178                                 pthread_spin_unlock(&(curr->lock));
179                                 pthread_spin_unlock(&(pred->lock));
180                                 return ret;
181                         }
182                         else {
183                                 pthread_spin_unlock(&(curr->lock));
184                                 pthread_spin_unlock(&(pred->lock));
185                                 return ret;
186                         }
187                 }
188         pthread_spin_unlock(&(pred->lock));
189                 pthread_spin_unlock(&(curr->lock));
190         }
191 
192 }
193 
194 /**
195  * Print a linked list.
196  **/
197 void ll_print(ll_t *ll)
198 {
199     ll_node_t *curr = ll->head;
200     printf("LIST [");
201     while (curr) {
202         if (curr->key == INT_MAX)
203             printf(" -> MAX");
204         else
205             printf(" -> %d", curr->key);
206         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;
    pthread_spinlock_t lock;
    int marked;
} ll_node_t;

struct linked_list {
    ll_node_t *head;
};

/**
 * 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;
    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);
```

```
52      ret->head = ll_node_new(-1);
53      ret->head->next = ll_node_new(INT_MAX);
54      ret->head->next->next = NULL;
55
56      return ret;
57  }
58
59  /**
60   * Free a linked list and all its contained nodes.
61   **/
62  void ll_free(ll_t *ll)
63  {
64      ll_node_t *next, *curr = ll->head;
65      while (curr) {
66          next = curr->next;
67          ll_node_free(curr);
68          curr = next;
69      }
70      XFREE(ll);
71  }
72
73  int ll_validate(ll_node_t *pred, ll_node_t *curr)
74  {
75      return (!pred->marked && !curr->marked && pred->next == curr);
76  }
77
78  int ll_contains(ll_t *ll, int key)
79  {
80      ll_node_t *curr;
81      curr = ll->head;
82
83      while (curr->key < key)
84          curr = curr->next;
85
86      return (curr->key == key && !curr->marked);
87
88  }
89
90  int ll_add(ll_t *ll, int key)
91  {
92          int ret = 0;
93
94          while(1) {
95                  ll_node_t *pred, *curr;
96                  ll_node_t *new_node;
97
98                  pred = ll->head;
99                  curr = pred->next;
100
101                 while (curr->key < key) {
102                         pred = curr;
103                         curr = curr->next;
104                 }
105                 pthread_spin_lock(&(pred->lock));
106                 pthread_spin_lock(&(curr->lock));
107
108                 if (ll_validate(pred, curr)){
109                         if (curr->key == key) {
```

```
                                pthread_spin_unlock(&(pred->lock));
                                pthread_spin_unlock(&(curr->lock));
                                return ret;
                        }
                        else {
                                ret = 1;
                                new_node = ll_node_new(key);
                                new_node->next = curr;
                                pred->next = new_node;
                                pthread_spin_unlock(&(pred->lock));
                                pthread_spin_unlock(&(curr->lock));
                                return ret;
                        }
                }
        pthread_spin_unlock(&(pred->lock));
                pthread_spin_unlock(&(curr->lock));
        }

}


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

        while(1) {
                ll_node_t *pred, *curr;
                ll_node_t *new_node;

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

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

                if (ll_validate(pred, curr)){
                        if (curr->key == key) {
                                ret = 1;
                curr->marked = 1;
                                pred->next = curr->next;
                                pthread_spin_unlock(&(pred->lock));
                                pthread_spin_unlock(&(curr->lock));
                                return ret;
                        }
                        else {
                                pthread_spin_unlock(&(pred->lock));
                                pthread_spin_unlock(&(curr->lock));
                                return ret;
                        }
                }
        pthread_spin_unlock(&(pred->lock));
                pthread_spin_unlock(&(curr->lock));
        }

```

```
168 }
169
170
171 /**
172  * Print a linked list.
173  **/
174 void ll_print(ll_t *ll)
175 {
176     ll_node_t *curr = ll->head;
177     printf("LIST [");
178     while (curr) {
179         if (curr->key == INT_MAX)
180             printf(" -> MAX");
181         else
182             printf(" -> %d", curr->key);
183         curr = curr->next;
184     }
185     printf(" ]\n");
186 }
```

- **Non blocking**

  Δεν καταφέραμε να υλοποίησουμε την συγκεκριμένη μέθοδο σωστά.

2. **Εκτελέστε την εφαρμογή για όλες τις διαφορετικές υλοποιήσεις λίστας. Εκτελέστε για 1, 2, 4, 8, 16, 32, 64 νήματα, για λίστες μεγέθους 1024 και 8192 και για συνδυασμούς λειτουργιών 80-10-10 και 20-40-40. Παρουσιάστε τα αποτελέσματά σας σε διαγράμματα και εξηγήστε την συμπεριφορά της εφαρμογής για κάθε κλείδωμα.**
   Σημείωση: σε όλες τις εκτελέσεις θα θέσετε κατάλληλα την μεταβλητή περιβάλλοντος MT_CONF ώστε τα νήματα να καταλαμβάνουν διαδοχικούς πυρήνες, π.χ. τα 16 νήματα εκτελούνται στους πυρήνες 0-15.

   - **Για συνδυασμό λειτουργιών 80-10-10**

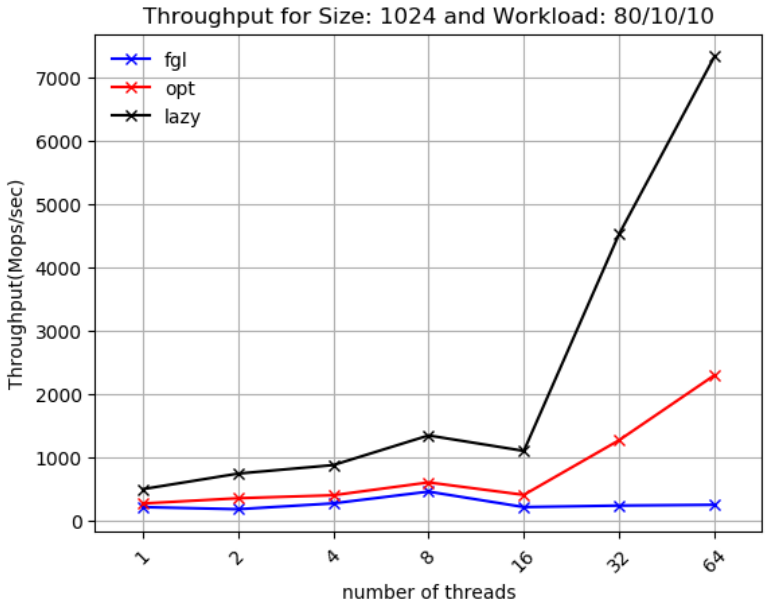


Figure 9: Throughput σε λίστα μεγέθους 1024 όταν έχουμε περισσότερες αναζητήσεις

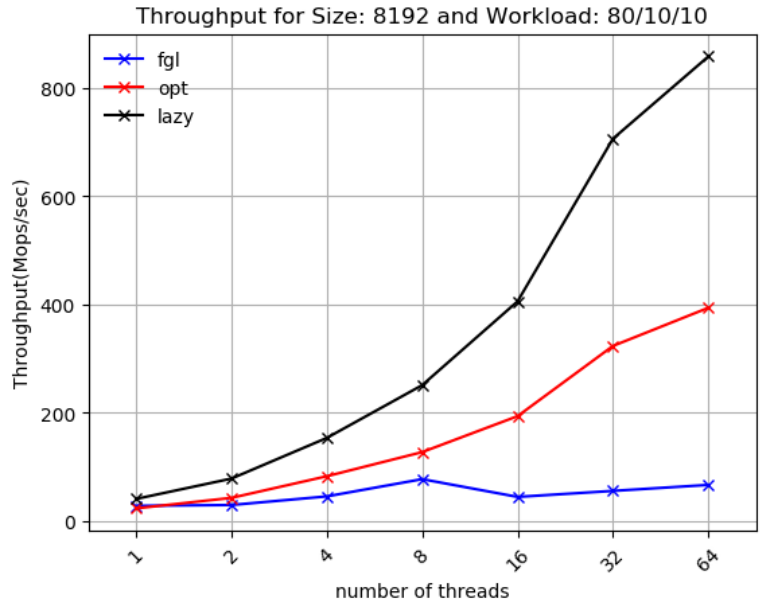


Figure 10: Throughput σε λίστα μεγέθους 8192 όταν έχουμε περισσότερες αναζητήσεις

- **Για συνδυασμό λειτουργιών 20-40-40**

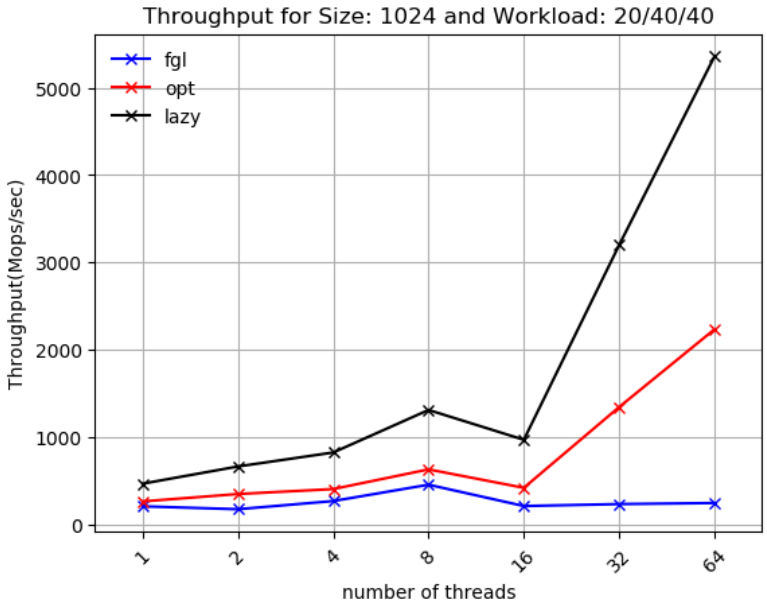


Figure 11: Throughput σε λίστα μεγέθους 1024 όταν έχουμε περισσότερες εισαγωγές-διαγραφές

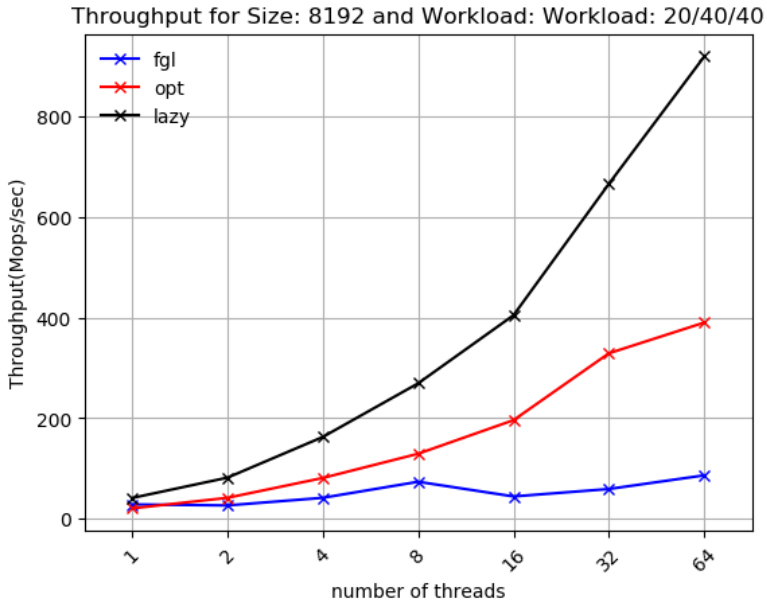


Figure 12: Throughput σε λίστα μεγέθους 8192 όταν έχουμε περισσότερες εισαγωγές-διαγραφές

Σχολιασμός :

(a) Αρχικά, παρατηρούμε ότι επιβεβαιώνεται η κατάταξη που είχαμε θεωρητικά στις διάφορες υλοποιήσεις. Δηλαδή, η optimistic υλοποίηση είναι καλύτερη από την fine grain και η lazy υλοποίηση είναι καλύτερη από όλες (ανεξάρτητα από το μέγεθος της λίστας και το μείγμα των λειτουργιών).

(b) Όσο αυξάνονται τα νήματα, η διαφορά τους μεγαλώνει καθώς έχουμε όλο και περισσότερες λειτουργίες και όλο και περισσότερες καταστάσεις συναγωνισμού.

(c) Ως προς το μέγεθος της λίστας, προφανώς όταν μεγαλώνει το μέγεθός της, το throughput πέφτει καθώς έχουμε να διατρέξουμε μία μεγαλύτερη λίστα.

(d) Ως προς το μείγμα των λειτουργιών, έχουμε σημαντική διαφορά στη μικρή λίστα για το lazy, όπου η επίδοση είναι καλύτερη όταν έχουμε κυρίως αναζητήσεις. Αυτό συμβαίνει, γιατί στη lazy υλοποίηση η contains δεν έχει κανένα κλείδωμα πράγμα που ευνοεί κατά πολύ την συνολική επίδοση.

Συνολικά, όπως ήταν αναμενόμενο η fgl υλοποίηση δεν κλιμακώνει καθόλου καθώς για κάθε λειτουργία έχει μια μεγάλη σειρά από λήψεις και απελευθερώσεις κλειδωμάτων και δεν επιτρέπει τα νήματα να εργαστούν ταυτόχρονα σε διαφορετικά σημεία. Αυτό βελτιώνεται με την optimized υλοποίηση ειδικότερα για πολλά νήματα καθώς μπορεί διατρέχουμε περισσότερες φορές την λίστα αλλά το κάνουμε χωρίς κλειδώματα. Τέλος, η lazy υλοποίηση είναι η καλύτερη καθώς με την βοήθεια των boolean μεταβλητών, το κλείδωμα γίνεται για πολύ μικρότερο χρόνο (ενώ στην contains δεν γίνεται καθόλου) ενώ επιτρέπεται στα νήματα να δουλεύουν παράλληλα σε διαφορετικές περιοχές. Η non blocking υλοποίηση δεν έγινε αλλά υποθέτουμε ότι θα ήταν η αποδοτικότερη από όλες καθώς δεν χρησιμοποιεί καθόλου κλειδώματα.

Αναφορές

[1] "Σημειώσεις του μαθήματος" http://www.cslab.ntua.gr/courses/pps/notes.go