Κουτρούμπας Αθανάσιος 03116073
Αραπίδης Μανώλης 03116071

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ
www.cslab.ece.ntua.gr

ΣΥΣΤΗΜΑΤΑ ΠΑΡΑΛΛΗΛΗΣ ΕΠΕΞΕΡΓΑΣΙΑΣ
3η ΑΣΚΗΣΗ

Θέματα Συγχρονισμού σε Σύγχρονα Πολυπύρηνα Συστήματα
Ακ. έτος 2020-2021, 9ο Εξάμηνο, Σχολή ΗΜ&ΜΥ

ΖΗΤΟΥΜΕΝΑ

2. Λογαριασμοί Τράπεζας

2.1.1

Υπάρχει ανάγκη για συγχρονισμό ανάμεσα στα νήματα της εφαρμογής;

Δεν υπάρχει ανάγκη για συγχρονισμό ανάμεσα στα νήματα, καθώς δεν υπάρχει ταυτόχρονη πρόσβαση των νημάτων σε κοινά δεδομένα. Το κάθε νήμα, επεξεργάζεται το δικό του κελί του πίνακα `accounts` σύμφωνα με το id του.

2.1.2

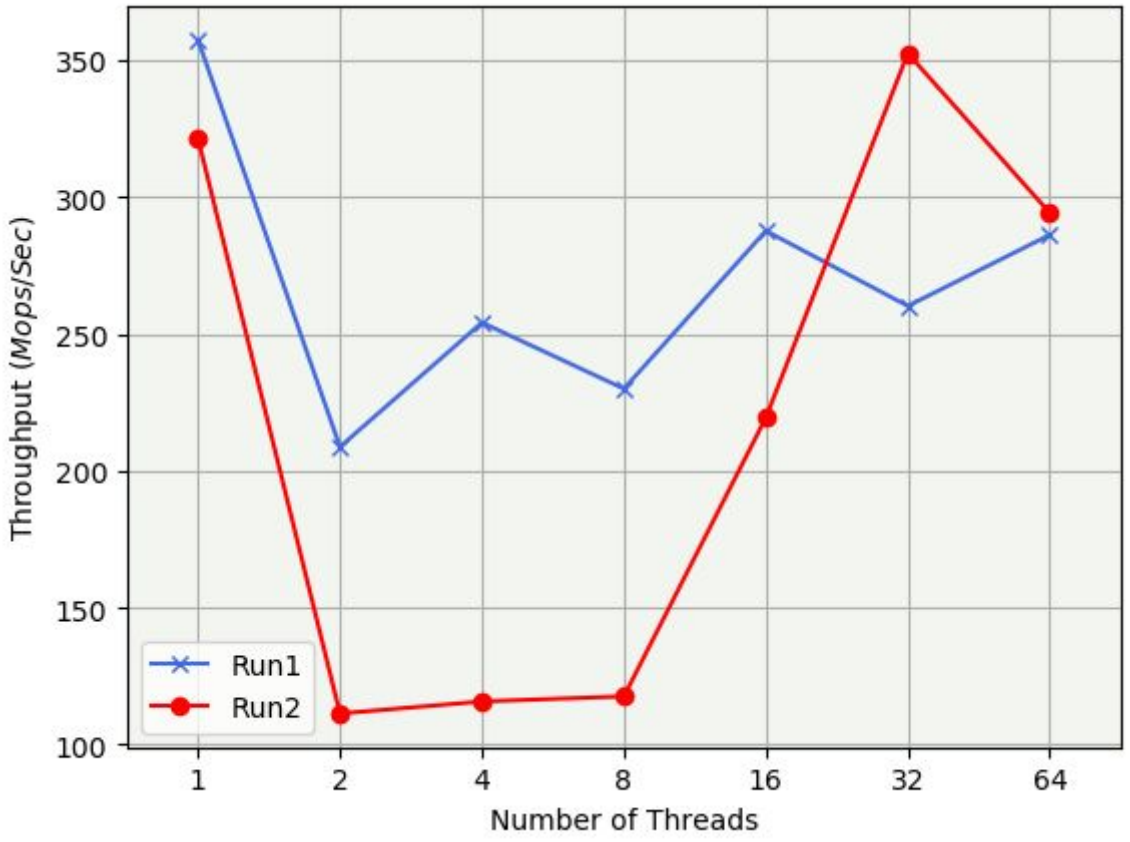
Πώς περιμένετε να μεταβάλλεται η επίδοση της εφαρμογής καθώς αυξάνετε τον αριθμό των νημάτων;

Όσο αυξάνεται ο αριθμός νημάτων της εφαρμογής αυξάνεται και ο αριθμός των κελιών που μπορεί να επεξεργάζεται παράλληλα. Το κάθε thread κάνει τον ίδιο αριθμό operations, οπότε αναμένουμε ότι θα αυξηθεί το throughput καθώς στον ίδιο χρόνο έχουμε περισσότερα operations.

2.1.3

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

Διαγράμματα Throughput



Αρχικά παρατηρούμε και για τις δύο εκτελέσεις, ότι δεν υπάρχει αύξηση του throughout όπως αναμέναμε κατά την αύξηση των threads.

Στην 1η εκτέλεση, τα νήματα προσαρμόζονται στον έναν επεξεργαστή (CPU 0) για τις εκτελέσεις για 1-16 threads. Σε αυτές τις εκτελέσεις υπάρχει αρχικά μείωση του throughput για περισσότερα από 1 thread και μετά υπάρχει μια κάπως αυξητική πορεία μέχρι τα 16 threads. Μετά για τα 32 και 64 threads, όπου έχουμε και εκτέλεση και στους άλλους 3 CPU (CPU 1,2,3) δεν έχουμε καμία αύξηση του throughput. Αυτό ίσως οφείλεται στην NUMA αρχιτεκτονική του συστήματος, καθώς πλέον έχουμε προσβάσεις και σε πιο απομακρυσμένες RAM.

Στην 2η εκτέλεση, τα νήματα προσαρμόζονται σε παραπάνω από έναν επεξεργαστή για τις εκτελέσεις για 1-64 threads. Έχουμε αντίστοιχα την ίδια συμπεριφορά μέχρι τα 8 threads, με πολύ χειρότερο throughput από ότι πριν. Όμως από τα 8 μέχρι τα 32 threads έχουμε αυξητική πορεία του throughput και ξεπερνάει τις τιμές της προηγούμενης εκτέλεσης.

Οι δύο εκτελέσεις, στο μόνο που διαφοροποιούνται είναι ότι η 1η προσπαθεί να τρέξει όσο περισσότερα thread μπορεί σε 1 CPU ενώ η 2η να ισομοιράσει τα threads και στις 4 CPU όπου μπορεί. Στην πρώτη περίπτωση αξιοποιούμε το locality που μας προσφέρει η NUMA αρχιτεκτονική, μεταξύ των threads και τις προσβάσεις στην ιεραρχία μνήμης, αλλά έχουμε το μειονέκτημα ότι για πάνω από 8 threads (έως 16 thread) σε μία CPU γίνεται χρήση του Hyperthreading, το οποίο δεν πρόκειται για αμιγώς παράλληλη εκτέλεση αλλά για concurrent εκτέλεση 2 threads. Οπότε για πάνω από 8 threads, η 2η εκτέλεση μπορεί να προσφέρει καλύτερες επιδόσεις, όμως για λιγότερα από 8 threads, οι επιδόσεις είναι χειρότερες λόγω της μη αξιοποίησης του locality των μνημών.

2.1.4

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;

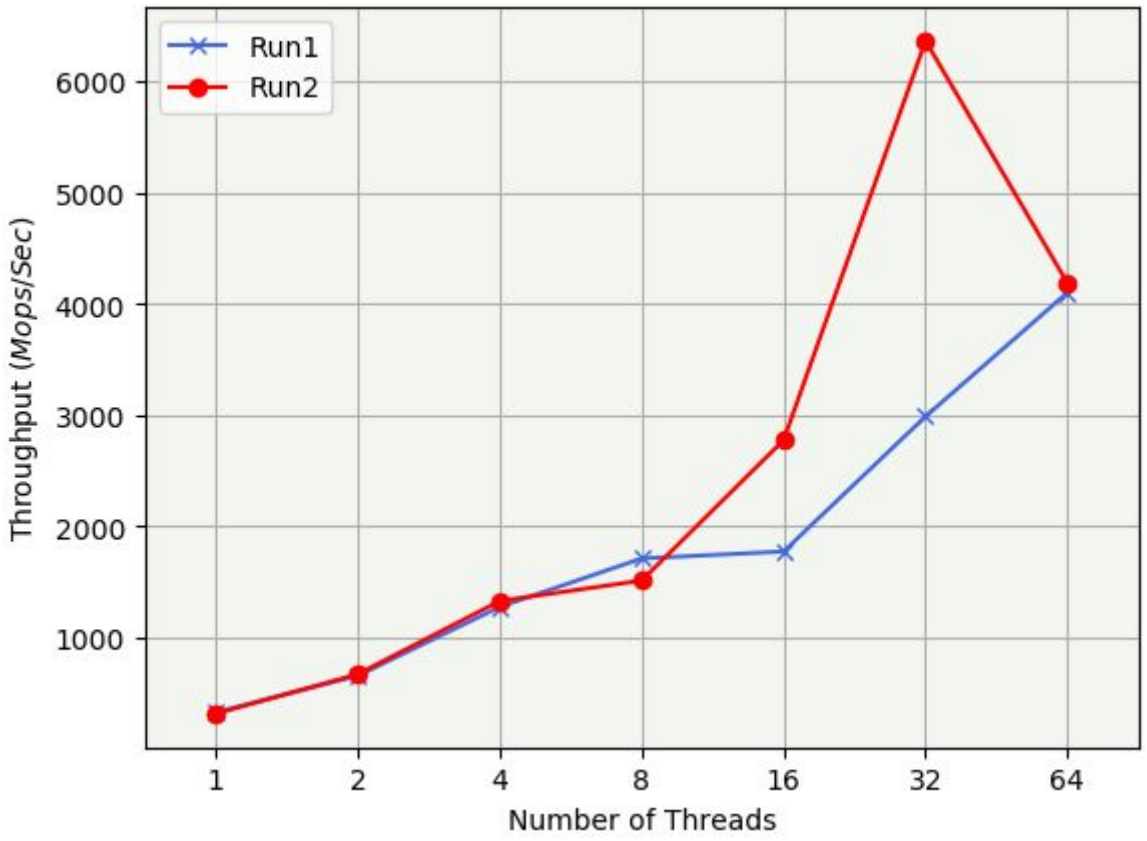
Άσχετα από τις διαφορές των δύο εκτελέσεων, το βασικό πρόβλημα που δεν έχουμε κλιμάκωση της εφαρμογής, είναι ότι το κάθε thread όταν θέλει να επεξεργαστεί το κελί του, φορτώνει και άλλα κελιά του πίνακα `accounts` στην cache του. Πιο συγκεκριμένα όσα στοιχεία του πίνακα, χωράνε στην cache line (που στην περίπτωση του Intel Xeon E5-4620 είναι 64 byte), οπότε αν κάποιο από τα άλλα στοιχεία έχει αλλαχθεί από ένα άλλο thread, η cache line δηλώνεται ως dirty, και λόγω του cache coherence πρωτοκόλλου θα πρέπει να ενημερωθεί πρώτα ώστε να προχωρήσει στην επεξεργασία του στοιχείου του το κάθε thread.

Μία λύση για το πρόβλημα αυτό είναι το κάθε thread, να φορτώνει στην cache μόνο το στοιχείο του και όχι τα υπόλοιπα, ώστε να μην υπάρχουν conflicts. Αυτό μπορεί να γίνει βάζοντας κατάλληλο padding στο struct του `accounts`, ώστε να γεμίζει η cache line μόνο με το στοιχείο του κάθε thread.
Πιο συγκεκριμένα, όπως υπάρχει και το padding για το struct `tdata_t`, έτσι υλοποιήσαμε και για το `accounts`:

```
struct {
    unsigned int value;
    char padding[64 - sizeof(int)];
} accounts[MAX_THREADS];
```

Οπότε ξανατρέχουμε με το καινούριο πρόγραμμα και το διάγραμμα που παίρνουμε είναι το παρακάτω:

Διαγράμματα Throughput (Βελτιωμένο)



Οπότε παρατηρούμε ότι πλέον, έχουμε ανάλογη αύξηση του throughput με κάθε διπλασιασμό των threads όπως αναμέναμε. Επίσης, η 2η εκτέλεση έχει πολύ καλύτερη επίδοση μετά τα 8 threads, γιατί όπως αναφέραμε η εκτέλεση των threads γίνεται σε ξεχωριστούς πυρήνες, σε αντίθεση με την 1η εκτέλεση που έχουμε 2 thread ανα core.

3. Αμοιβαίος Αποκλεισμός - Κλειδώματα

3.1.1

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

Αρχικά έχουμε την υλοποίηση του TTAS κλειδώματος, το οποίο είναι σαν το TAS κλείδωμα, απλά πρώτα περιμένουμε να γίνει Unlocked το κλείδωμα και μετά μπαίνουμε στην διαδικασία του test-and-set.

`ttas_lock:`

```
#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);
    lock->state = UNLOCKED;
    return lock;
}

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

void lock_acquire(lock_t *lock)
{
    lock_t *l = lock;
    while(1){
        while(l->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);
}
```

Έπειτα έχουμε την υλοποίηση του array κλειδώματος, η οποία είναι ίδια με αυτή στις διαλέξεις του μαθήματος. Χρησιμοποιούμε την `__sync_fetch_and_add`, ώστε όταν ένα thread πέρνει το lock να μπορεί ατομικά να διαβάσει την τιμή του tail και να την αυξήσει κατά 1.

array_lock:

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

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

__thread int thread_slot;

lock_t *lock_init(int nthreads)
{
    lock_t *lock;
    XMALLOC(lock,1);
    XMALLOC(lock->flag,nthreads);
    lock->size=nthreads;
    lock->tail=0;
    lock->flag[0]=true;
    return lock;
}

void lock_free(lock_t *lock)
{
    XFREE(lock->flag);
    XFREE(lock);
}

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

void lock_release(lock_t *lock)
{
    lock->flag[thread_slot]=false;
    lock->flag[(thread_slot+1)%lock->size]=true;
}
```

Έπειτα έχουμε την υλοποίηση του κλειδώματος με pthread spinlocks.
Στην `lock_init()` καλούμε την `pthread_spin_init()`, ώστε να κάνει τις απαραίτητες αρχικοποιήσεις και δεσμεύσεις πόρων για το spinlock και να το αρχικοποιήσει σε κατάσταση Unlocked.
Στην `lock_acquire()` καλούμε την `pthread_spin_lock()`, ώστε να πάρει το thread το spinlock. Σε περίπτωση που έχει κρατηθεί από άλλο thread, τότε αυτό κάνει busy-wait μέχρι το lock να γίνει διαθέσιμο.
Στην `lock_release()` καλούμε την `pthread_spin_unlock()`, ώστε το thread να μπορέσει να αφήσει το spinlock.

pthread_lock:

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

struct lock_struct {
    pthread_spinlock_t spin_lock;
};

lock_t *lock_init(int nthreads)
{
    int ret;
    lock_t *lock;
    XMALLOC(lock, 1);
    ret=pthread_spin_init(&lock->spin_lock,PTHREAD_PROCESS_PRIVATE);
    if(!ret)
        return lock;
    else if(ret==EAGAIN){
        fprintf(stderr, "Out of memory: %s:%d\n", __FILE__, __LINE__);
        exit(1);
    }
    else if(ret==EBUSY){
        fprintf(stderr, "Is already iniatialized: %s:%d\n", __FILE__, __LINE__);
        exit(1);
    }
    else if(ret==EINVAL){
        fprintf(stderr, "Invalid lock: %s:%d\n", __FILE__, __LINE__);
        exit(1);
    }
    return NULL;
}

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

void lock_acquire(lock_t *lock)
{
    int ret;
    while(1){
        ret = pthread_spin_lock(&lock->spin_lock);
```

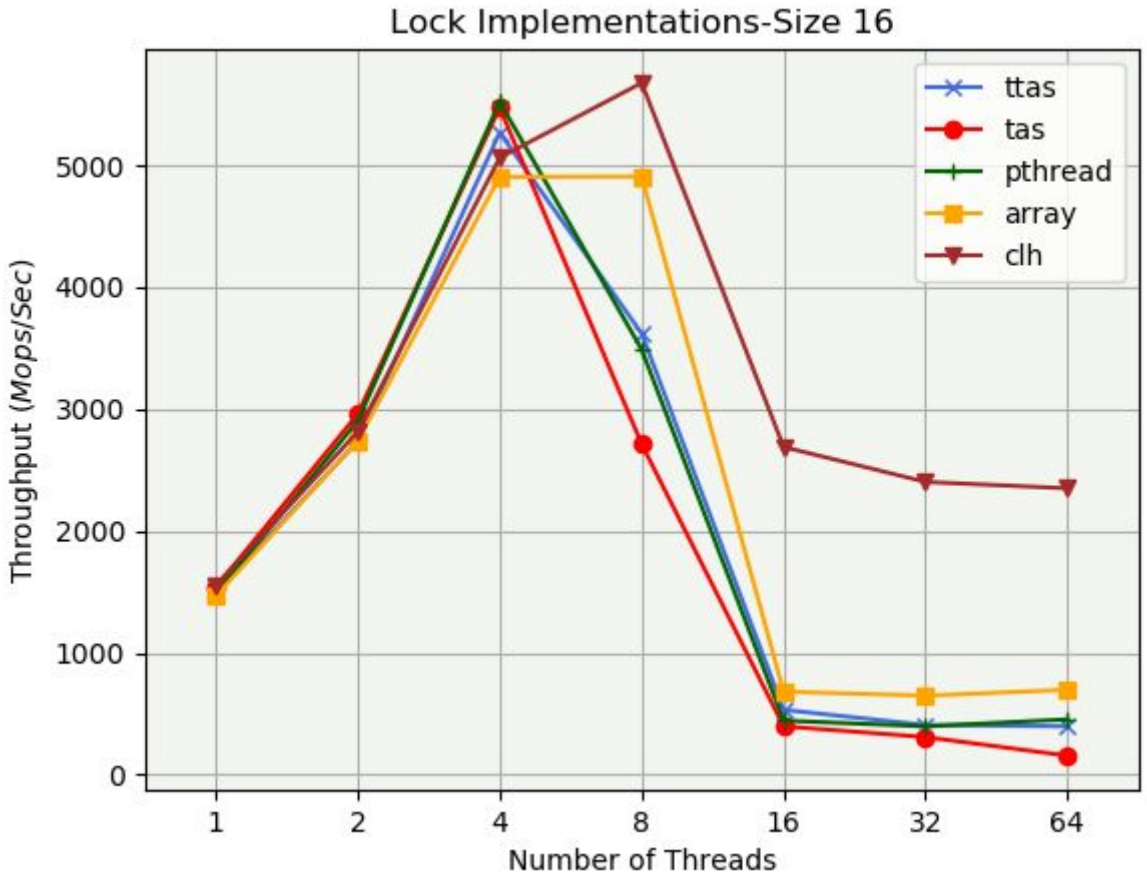
```
        if(ret==0) break;
        else if(ret==EDEADLK) break;
        else if(ret==EINVAL){
                fprintf(stderr, "Uninitialized spinlock: %s:%d\n", __FILE__, __LINE__);
                exit(1);
        }
    };
}

void lock_release(lock_t *lock)
{
    int ret;
    ret=pthread_spin_unlock(&lock->spin_lock);
    if(ret==0)
        return;
    else if(ret==EPERM){
        fprintf(stderr, "Thread does not hold the spinlock: %s:%d\n", __FILE__, __LINE__);
                exit(1);
    }
    else if(ret==EINVAL){
                fprintf(stderr, "Uninitialized spinlock: %s:%d\n", __FILE__, __LINE__);
                exit(1);
    }
}
```

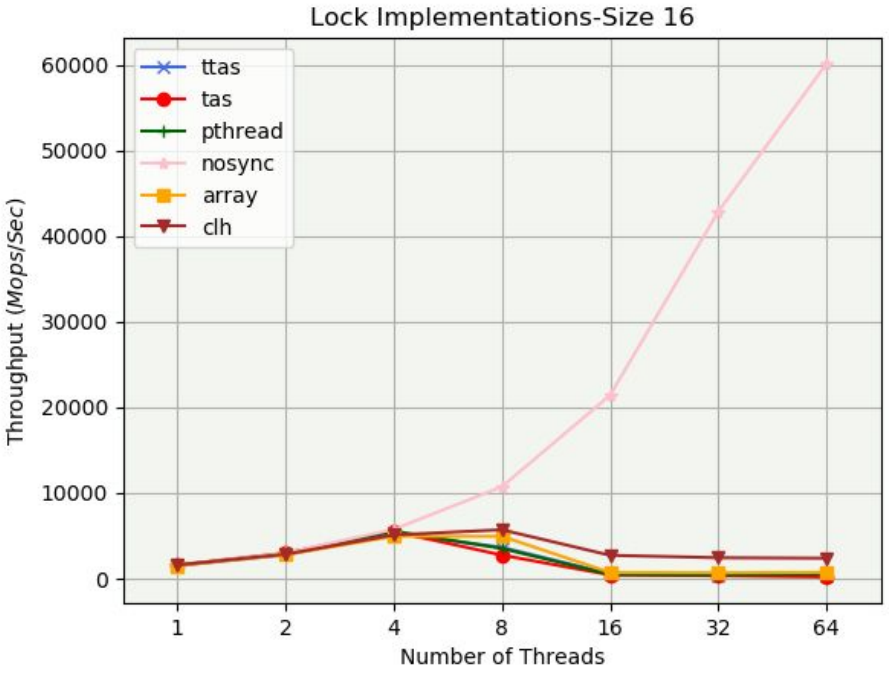
3.1.2

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

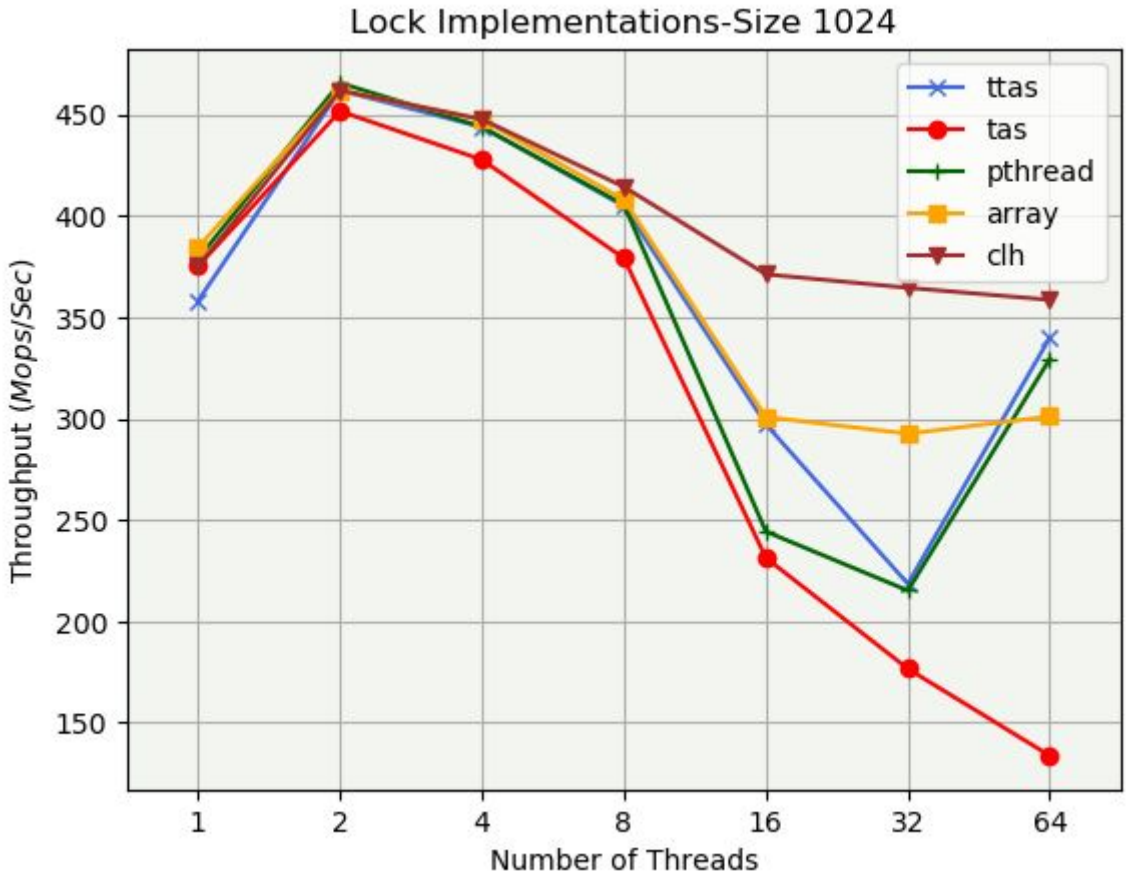
Εκτελέσαμε τις μετρήσεις και από τα αποτελέσματα που πήραμε δημιουργήσαμε τα παρακάτω διαγράμματα. Για κάθε μέγεθος έχουμε 2 διαγράμματα, ένα με και ένα χωρις την υλοποίηση του nosync lock, καθώς οι επιδόσεις του έχουν τεράστιες διαφορές από τα υπόλοιπα, οπότε χάνεται η ακρίβεια μεταξύ των άλλων. Το συμπεριλαμβάνουμε όμως, για να φαίνεται ένα άνω όρια που θα μπορούσαμε να έχουμε στην ιδανική περίπτωση μη χρησιμοποίησης των κλειδωμάτων.

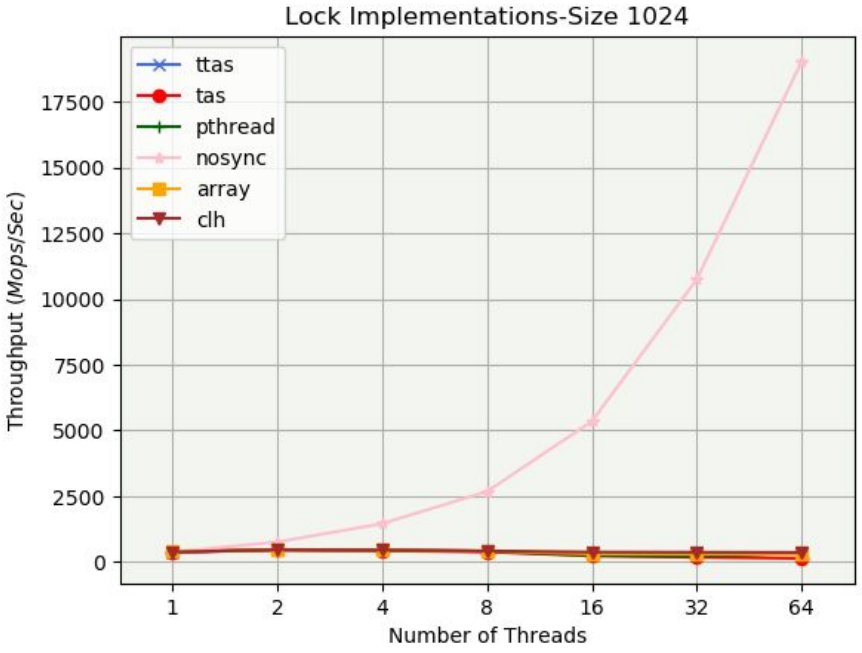
Διαγράμματα Throughput (Size = 16)



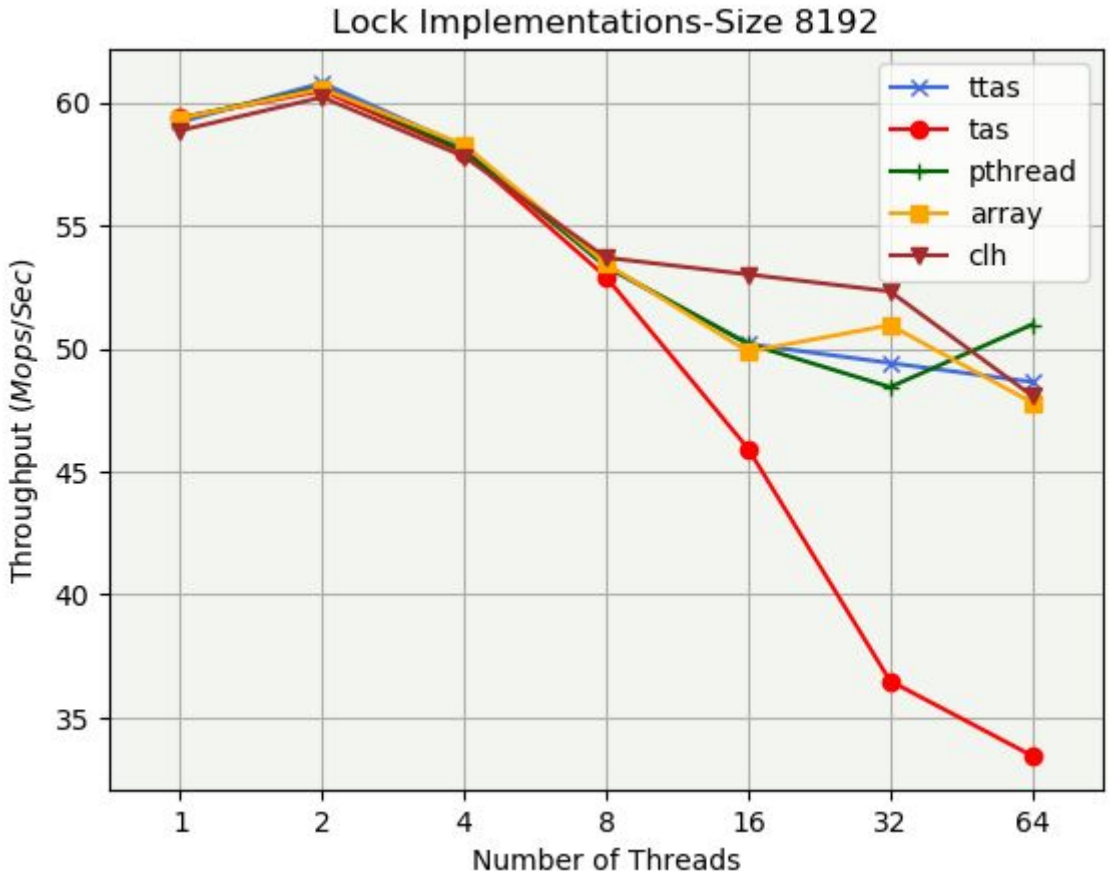


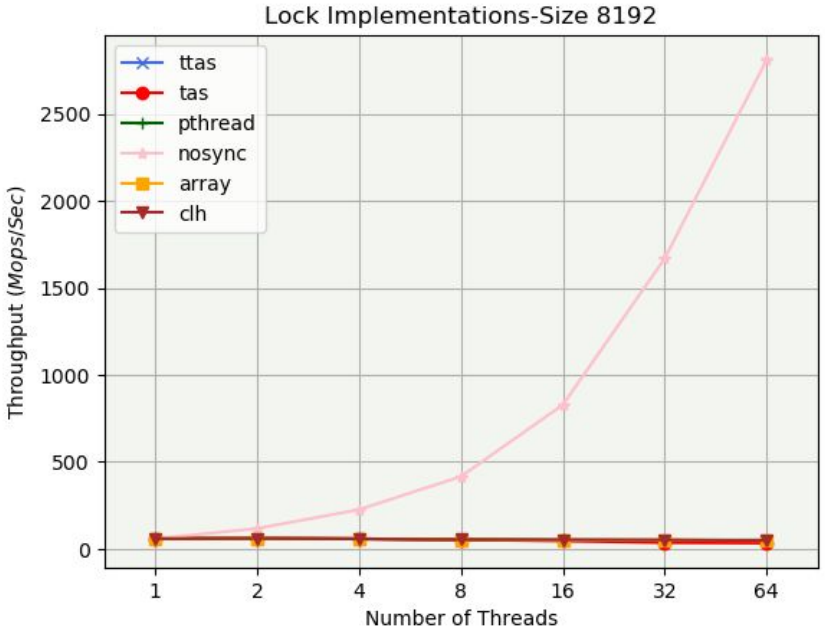
Διαγράμματα Throughput (Size = 1024)





Διαγράμματα Throughput (Size = 8192)





Συμπεράσματα:

Αρχικά, παρατηρώντας τα διαγράμματα για όλα τα μεγέθη, βλέπουμε ότι στην υλοποίηση χωρίς κλειδώματα έχουμε σχεδόν εκθετική αύξηση του throughput όσο αυξάνονται τα threads, ενώ τα υπόλοιπα συγκριτικά φαίνεται να έχουν σταθερό throughput (ή και χειρότερο όπως για το μικρότερο μέγεθος). Προφανώς όμως η υλοποίηση χωρίς κλειδώματα είναι μη ορθή, αλλά μας τα αποτελέσματα μας δίνουν μια αίσθηση για το πόσο επιβαρύνουν τα κλειδώματα την απόδοση μιας εφαρμογής.

Επίσης για όλες σχεδόν τις υλοποιήσεις, βλέπουμε ότι υπάρχει μια πορεία μείωσης του throughput για μεγάλο αριθμό threads (32 και 64). Αυτό συμβαίνει μάλλον, γιατί καθώς αυξάνονται τα threads, αυξάνονται και διεκδικήσεις για τα locks, οπότε στην περίπτωση αυτής της εφαρμογής τα περισσότερα threads μας προσφέρουν μεγαλύτερο overhead παρά αύξηση του throughput.

Τώρα αναλυτικά για την κάθε υλοποίηση:

Για το **tas** και **ttas** lock, βλέπουμε ότι υπάρχει σαφώς βελτίωση του δεύτερου σε σχέση με του πρώτου, και αυτή η διαφορά γίνεται πιο αισθητή καθώς αυξάνεται το μέγεθος της λίστας, όσο και ο αριθμός των threads για κάθε μέγεθος. Ειδικότερο το **tas**, έχει την χειρότερη απόδοση σχεδόν από όλες τις υλοποιήσεις. Και στις δύο υλοποιήσεις, υπάρχει πτώση του throughput με την αύξηση των threads, όμως για το **tas** lock είναι συνεχώς φθίνουσα η καμπύλη, ενώ το throughput του **ttas** φαίνεται να σταθεροποιείται για παραπάνω από 16 threads. Αυτό όπως έχουμε δει και στην θεωρία είναι λογικό, καθώς το **ttas** είναι μια βελτίωση του **tas**, η οποία δεν προκαλεί συμφόρηση του bus λόγω του cache coherence πρωτοκόλλου (συνεχώς εγγραφές στο state, που κάνουν invalid το cache block), αλλά διαβάζοντας πρώτα το state αν γίνει Unlocked και μετά κάνοντας την λειτουργία του test-and-set.

Για το **pthread spinlock**, βλέπουμε ότι έχει παρόμοιες αποδόσεις με το **ttas** lock. Πρόκειται για ένα από τα πιο απλά lock, καθώς το κάθε thread απλά περιμένει μέχρι να γίνει unlocked το lock για να το πάρει αυτό, και για τα μεγέθη λιστών και threads που έχουμε φαίνεται να έχει αρκετά ικανοποιητικές επιδόσεις σε σύγκριση με τα υπόλοιπα.

Για το **array** lock, επίσης έχει παρόμοια επίδοση με το **pthread spinlock** και το **ttas** lock, όμως για μέγεθος 1024, έχει λίγο πιο καλή επίδοση για 16 και 32 threads. Στο συγκεκριμένο κλείδωμα λόγω της χρήσης πίνακα, μπορεί να έχουμε εμφάνιση του φαινομένου του false sharing (όπως και στην περίπτωση του **tas**), όπου ένα cache block μπορεί να γίνει invalid και να έχουμε πολύ traffic λόγω του cache coherence πρωτοκόλλου. Αυτό βέβαια λύνεται προσθέτοντας ένα κατάλλο padding ώστε το κάθε thread να έχει τοπικά μόνο το δικό του στοιχείο του πίνακα που θα γράφει και διαβάζει. Όμως έτσι αυξάνονται οι απαιτήσεις σε μνήμη που χρειάζεται το κλείδωμα.

Τέλος, το καλύτερο σχεδόν σε όλες τις περιπτώσεις κλείδωμα είναι το **clh** lock, το οποίο ειδικά στα μικρότερα μεγέθη και στα πολλά threads, έχει πολύ μεγαλύτερο throughput από όλα τα υπόλοιπα.

4. Τακτικές συγχρονισμού για δομές δεδομένων

4.1.1

Υλοποιήστε τις ζητούμενες λίστες συμπληρώνοντας τα αντίστοιχα αρχεία της μορφής ll_<sync_type>.c.

Παρακάτω έχουμε τις απαιτούμενες υλοποιήσεις όλων των τακτικών για τις ταξινομημένες συνδεδεμένες λίστες. Για κάθε τακτική συγχρονισμού, υλοποιήσαμε τις συναρτήσεις `ll_add()`, `ll_remove()`, `ll_contains()` και συμπληρώσαμε όπου χρειαζόταν τις `ll_node_new()`, `ll_node_free()`.

Fine-grain Locking:

Για αυτήν την τακτική, προσθέσαμε στο struct `ll_node` ένα lock (χρησιμοποιώντας την έτοιμη `pthread_lock.c` υλοποίηση από το προηγούμενο ερώτημα). Η υλοποίηση είναι ίδια με αυτή που αναγράφεται και στις διαφάνειες της διάλεξης, όπου στην ουσία έχουμε hand-over-hand locking για όλες τις λειτουργίες add, remove, contains.

```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"
#include "lock.h"

typedef struct ll_node {
    int key;
    struct ll_node *next;
    lock_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->lock=lock_init(0);
    ret->key = key;
    ret->next = NULL;
    /* Other initializations here? */

    return ret;
}
```

```c
/**
 * Free a linked list node.
 **/
static void ll_node_free(ll_node_t *ll_node)
{
    lock_free(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)
{
    ll_node_t *prev, *curr;
    prev=ll->head;
    curr=prev->next;
    lock_acquire(prev->lock);
    lock_acquire(curr->lock);

    while(curr->key < key){
        lock_release(prev->lock);
        prev=curr;
        curr=curr->next;
        lock_acquire(curr->lock);
    }

    if(curr->key==key){
        lock_release(prev->lock);
        lock_release(curr->lock);
```

```
        return 1;
    }
    lock_release(prev->lock);
    lock_release(curr->lock);
    return 0;
}

int ll_add(ll_t *ll, int key)
{
    ll_node_t *prev, *curr, *new;
    prev=ll->head;
    curr=prev->next;
    lock_acquire(prev->lock);
    lock_acquire(curr->lock);

    while(curr->key < key){
        lock_release(prev->lock);
        prev=curr;
        curr=curr->next;
        lock_acquire(curr->lock);
    }

    if(curr->key==key){
        lock_release(prev->lock);
        lock_release(curr->lock);
        return 0;
    }
    else{
        new=ll_node_new(key);
        prev->next=new;
        new->next=curr;
    }
    lock_release(prev->lock);
    lock_release(curr->lock);
    return 1;
}

int ll_remove(ll_t *ll, int key)
{
    ll_node_t *prev, *curr;
    prev=ll->head;
    curr=prev->next;
    lock_acquire(prev->lock);
    lock_acquire(curr->lock);

    while(curr->key < key){
        lock_release(prev->lock);
        prev=curr;
        curr=curr->next;
        lock_acquire(curr->lock);
    }

    if(curr->key==key){
        prev->next=curr->next;
        lock_release(prev->lock);
        lock_release(curr->lock);
```

```
        // ll_node_free(curr);
        return 1;
    }
    lock_release(prev->lock);
    lock_release(curr->lock);
    return 0;
}

/**
 * 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:

Για αυτήν την τακτική, προσθέσαμε στο struct `ll_node` ένα pthread spinlock ως lock για τα κλειδώματα. Η υλοποίηση είναι ίδια με αυτή που αναγράφεται και στις διαφάνειες της διάλεξης, όπου στην ουσία έχουμε local locking για όλες τις λειτουργίες add, remove, contains, και retries για τις περιπτώσεις της add και remove. Για όλες τις λειτουργίες επίσης χρησιμοποιείται η validate, η οποία διατρέχει όλη την λίστα κάθε φορά.

```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"
#include "lock.h"

typedef struct ll_node {
    int key;
    struct ll_node *next;
    // lock_t *lock;
    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->lock=lock_init(0);
    pthread_spin_init(&ret->lock,PTHREAD_PROCESS_PRIVATE);
    ret->key = key;
    ret->next = NULL;
    /* Other initializations here? */

    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_validate(ll_t *ll, ll_node_t *prev, ll_node_t *curr){
    ll_node_t *node = ll->head;

    while(node->key<=prev->key){
        if(node==prev)
               return prev->next==curr;
        node=node->next;
    }
    return 0;
}

int ll_contains(ll_t *ll, int key)
{
    // printf("contains in\n");
    ll_node_t *prev, *curr;
    int flag=-1;
    while(1)
    {
        prev=ll->head;
        curr=prev->next;

        while(curr->key < key){
               prev=curr;
               curr=curr->next;
        }
        pthread_spin_lock(&prev->lock);
        pthread_spin_lock(&curr->lock);
        if(ll_validate(ll,prev,curr)){
```

```
                if(curr->key==key)
                        flag=1;
                else
                        flag=0;
        }
        pthread_spin_unlock(&prev->lock);
        pthread_spin_unlock(&curr->lock);
        if (flag==0 || flag==1)
                break;
    }
    // printf("contains out\n");
    return flag;
}

int ll_add(ll_t *ll, int key)
{
    // printf("add in\n");
    ll_node_t *prev, *curr, *new;
    int flag=-1;
    while(1)
    {
        prev=ll->head;
        curr=prev->next;

        while(curr->key < key){
                prev=curr;
                curr=curr->next;
        }
        pthread_spin_lock(&prev->lock);
        pthread_spin_lock(&curr->lock);
        if(ll_validate(ll,prev,curr)){
                if(curr->key==key)
                        flag=0;
                else{
                        new=ll_node_new(key);
                        new->next=curr;
                        prev->next=new;
                        flag=1;
                }
        }
        pthread_spin_unlock(&prev->lock);
        pthread_spin_unlock(&curr->lock);
        if (flag==0 || flag==1)
                break;
    }
    // printf("add out\n");
    return flag;
}


int ll_remove(ll_t *ll, int key)
{
    // printf("remove in\n");
    ll_node_t *prev, *curr;
    int flag=-1;
    while(1)
```

```
    {
        prev=ll->head;
        curr=prev->next;

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

        pthread_spin_lock(&prev->lock);
        pthread_spin_lock(&curr->lock);
        if(ll_validate(ll,prev,curr)){
                if(curr->key==key){
                        prev->next=curr->next;
                        flag=1;
                }
                else{
                        flag=0;
                }
        }
        // else
        //     continue;
        pthread_spin_unlock(&prev->lock);
        pthread_spin_unlock(&curr->lock);
        if(flag==1 || flag==0)
                break;
    }
    // printf("remove out\n");
    return flag;
}

/**
 * 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:

Για αυτήν την τακτική, προσθέσαμε πάλι στο struct `ll_node` ένα pthread spinlock ως lock για τα κλειδώματα, αλλά και ένα bool πεδίο `mark`, το οποίο είναι `true` αν ο κόμβος έχει διαγραφεί λογική και `false` αν υπάρχει. Η υλοποίηση είναι ίδια με αυτή που αναγράφεται και στις διαφάνειες της διάλεξης, όπου στην ουσία δεν χρειαζόμαστε locking για την λειτουργία του contains και έχουμε local locking και retries για τις λειτουργίες add και remove. Επίσης η validate, πλέον κάνει local έλεγχο τον κόμβων και δεν διατρέχει όλη την λίστα.

```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"
#include <stdbool.h>

typedef struct ll_node {
    int key;
    struct ll_node *next;
    pthread_spinlock_t lock;
    bool mark;
} 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);
    pthread_spin_init(&ret->lock,PTHREAD_PROCESS_PRIVATE);
    ret->key = key;
    ret->next = NULL;
    ret->mark = false;
    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);
}
```

```c
/**
 * 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_validate(ll_node_t *prev, ll_node_t *curr){
    return (!prev->mark && !curr->mark && prev->next==curr);
}

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

int ll_add(ll_t *ll, int key)
{
    ll_node_t *prev, *curr, *new;
    int flag=-1;
    while(1)
    {
        prev=ll->head;
        curr=prev->next;

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

```
        pthread_spin_lock(&prev->lock);
        pthread_spin_lock(&curr->lock);
        if(ll_validate(prev,curr)){
                if(curr->key==key)
                        flag=0;
                else{
                        new=ll_node_new(key);
                        new->next=curr;
                        prev->next=new;
                        flag=1;
                }
        }
        pthread_spin_unlock(&prev->lock);
        pthread_spin_unlock(&curr->lock);
        if (flag==0 || flag==1)
                break;
    }
    return flag;
}

int ll_remove(ll_t *ll, int key)
{
    ll_node_t *prev, *curr;
    int flag=-1;
    while(1)
    {
        prev=ll->head;
        curr=prev->next;

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

        pthread_spin_lock(&prev->lock);
        pthread_spin_lock(&curr->lock);
        if(ll_validate(prev,curr)){
                if(curr->key==key){
                        curr->mark=true;
                        prev->next=curr->next;
                        flag=1;
                }
                else{
                        flag=0;
                }
        }
        pthread_spin_unlock(&prev->lock);
        pthread_spin_unlock(&curr->lock);
        if(flag==1 || flag==0)
                break;
    }
    return flag;
}

/**
 * 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:

Για αυτήν την τακτική, δεν έχουμε πλέον lock και το πεδίο `mark` πλέον το ενσωματώνουμε στην διεύθυνση του επόμενου κόμβου (`struct ll_node *next`). Χρησιμοποιούμε το τελευταίο bit της διεύθυνσης ως marked, όπου αν 0 ο κόμβος υπάρχει και αν 1 ο κόμβος έχει διαγραφεί λογικά. Η αρχιτεκτονική του επεξεργαστή μας επιτρέπει να πειράξουμε το τελευταίο bit της διεύθυνσης χωρίς να επηρεαστεί αυτή, καθώς πάντα το τελευταίο bit μιας διευθυνσης είναι 0, λόγω alignment ανα 4 byte (για 32 bit) ή ανα 8 byte (για 64 bit).

Για την υλοποίηση έχουμε δημιουργήσει τις συναρτήσεις που υπάρχουν και στις διαφάνειες της διάλεξης, δηλαδή τις:

- `getReference(ll_node_t *node)`: Επιστρέφει την διεύθυνση του κόμβου, χωρίς το πεδίο marked.
- `get(ll_node_t *node, bool *mark)`: Επιστρέφει την διεύθυνση του κόμβου, και γυρνάει την τιμή του marked bit στο mark.
- `compareAndset(ll_node_t *node, ll_node_t *expectedRef, ll_node_t *updateRef, bool expectedMark, bool updateMark)`: Η τιμή της Expected και Update διεύθυνσης και mark, γίνονται λογικό or για να συνδυαστούν. Ελέγχει και αν επιτύχει ενημερώνει και τα δύο πεδία (διεύθυνσης και mark) χρησιμοποιώντας την `__sync_bool_compare_and_swap`, για ατομικό compare and swap.

Η υλοποίηση των υπολοίπων λειτουργιών είναι όπως και στις διαφάνειες Με αυτήν την τακτική δεν χρειαζόμαστε πλέον την validate και η add, remove γίνονται χωρίς lock με retries.

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

#include "../common/alloc.h"
#include "ll.h"
#include <stdbool.h>

typedef struct ll_node {
    int key;
    struct ll_node *next;
} ll_node_t;

struct linked_list {
    ll_node_t *head;
};

typedef struct window{
    ll_node_t *prev;
    ll_node_t *curr;
} window_t;

/**
 * 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? */

    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);
}

ll_node_t *getReference(ll_node_t *node)
{
    ll_node_t *next_ref;
    next_ref = (ll_node_t *) ((unsigned long long)node & (~1));
    return next_ref;
}

ll_node_t *get(ll_node_t *node, bool *mark)
```

```
{
    ll_node_t *next_ref;
    *mark = (bool) ((unsigned long long) node & 1);
    next_ref = (ll_node_t *) ((unsigned long long) node & (~1));
    return next_ref;
}


bool compareAndset(ll_node_t *node, ll_node_t *expectedRef, ll_node_t *updateRef,
                        bool expectedMark, bool updateMark)
{
    ll_node_t *expected, *update;
    bool ret;
    expected = (ll_node_t *) ((unsigned long long) expectedRef | expectedMark);
    update = (ll_node_t *) ((unsigned long long) updateRef | updateMark);
    ret = __sync_bool_compare_and_swap(&node,expected,update);
    return ret;
}

window_t *find(ll_t *ll, int key){
    ll_node_t *prev=NULL, *curr=NULL, *succ=NULL;
    window_t *ret;
    bool mark=false, snip;
    XMALLOC(ret,1);
retry:
    while(true){
        prev = ll->head;
        curr = getReference(prev->next);
        while(true){
                succ = get(curr->next,&mark);
                while(mark){
                        snip = compareAndset(prev->next, curr, succ, false, false);
                        if(!snip) goto retry;
                        curr = succ;
                        succ = get(curr->next, &mark);
                }
                if(curr->key >= key){
                        ret->prev = prev;
                        ret->curr = curr;
                        return ret;
                }
                prev = curr;
                curr = succ;
        }
    }
}

int ll_contains(ll_t *ll, int key)
{
    bool mark;
    ll_node_t *curr;
    curr = ll->head;
    while(curr->key < key)
        curr = getReference(curr->next);
    get(curr->next, &mark);
    return (curr->key == key && !mark);
```

```
}

int ll_add(ll_t *ll, int key)
{
    bool slice;
    ll_node_t *prev=NULL, *curr=NULL, *node=NULL;
    window_t *ret;
    while(true){
        ret = find(ll,key);
        prev = ret->prev;
        curr = ret->curr;
        XFREE(ret);
        if (curr->key == key){
                        return false;
                }
        else{
                node = ll_node_new(key);
                node->next = curr;
                if(compareAndset(prev->next, curr, node, false, false)){
                        return true;
                }
        }
    }
    return 0;
}

int ll_remove(ll_t *ll, int key)
{
    ll_node_t *prev=NULL, *curr=NULL, *succ=NULL;
    bool snip;
    window_t *ret;
    while(true){
        ret = find(ll,key);
        prev = ret->prev;
        curr = ret->curr;
        XFREE(ret);
        if (curr->key != key)
                return false;
        else{
                succ = getReference(curr->next);
                snip = compareAndset(curr->next, succ, succ, false, true);
                if(!snip)
                        continue;
                compareAndset(prev->next, curr, succ, false, false);
                return true;
        }
    }
    return 0;
}

/**
 * Print a linked list.
 **/
void ll_print(ll_t *ll)
{
    ll_node_t *curr = ll->head;
```

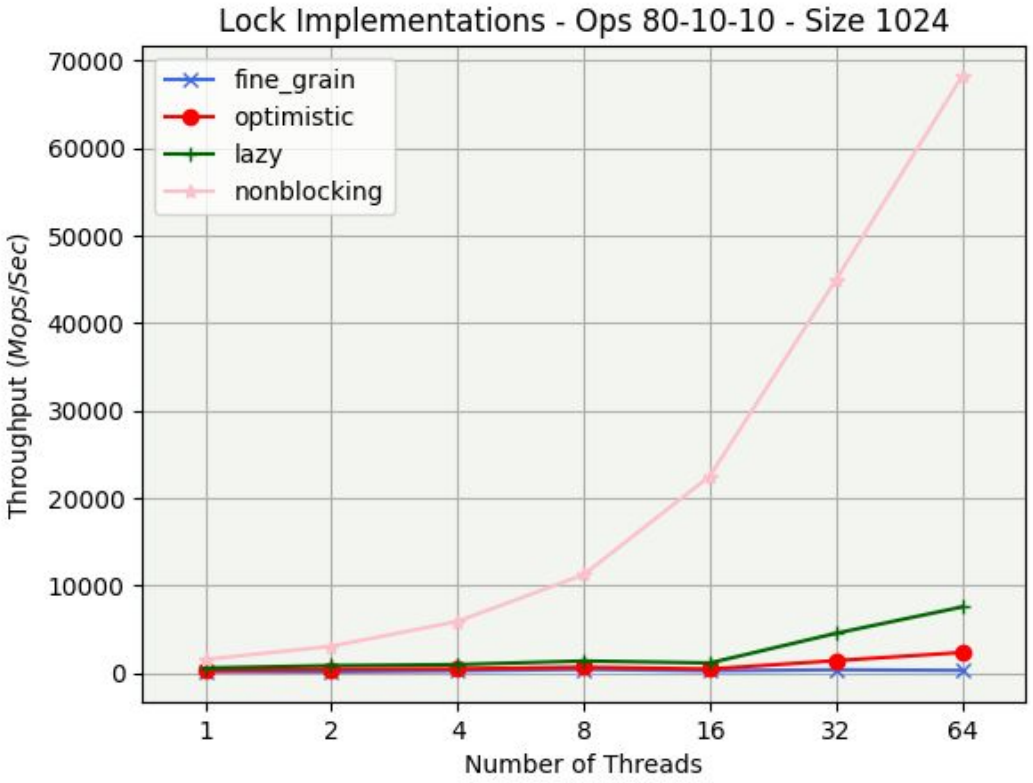
```
    printf("LIST [");
    while (curr) {
        if (curr->key == INT_MAX)
                printf(" -> MAX");
        else
                printf(" -> %d", curr->key);
        curr = curr->next;
    }
    printf(" ]\n");
}
```

4.1.2

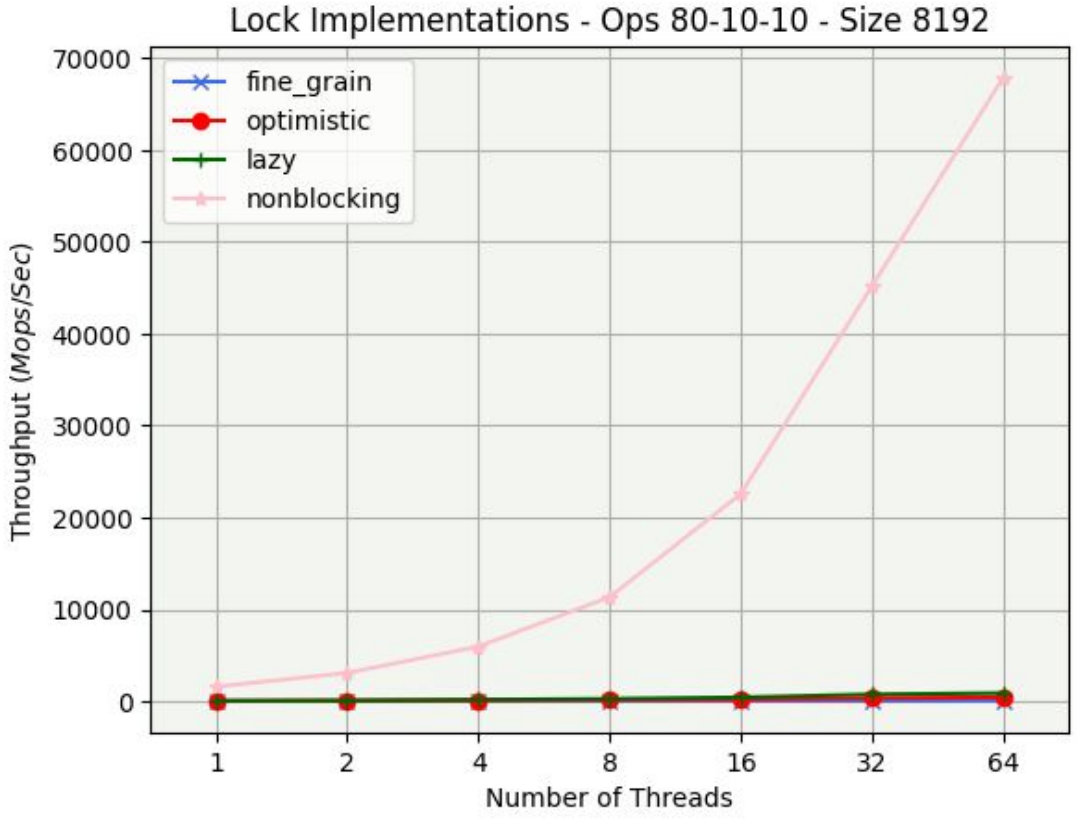
Εκτελέστε την εφαρμογή για όλες τις διαφορετικές υλοποιήσεις λίστας. Εκτελέστε για 1,2,4,8,16,32,64 νήματα, για λίστες μεγέθους 1024 και 8192 και για συνδυασμούς λειτουργιών 80-10-10 και 20-40-40. Παρουσιάστε τα αποτελέσματά σας σε διαγράμματα και εξηγήστε την συμπεριφορά της εφαρμογής για κάθε κλείδωμα. 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.

Εκτελέσαμε τις μετρήσεις και από τα αποτελέσματα που πήραμε δημιουργήσαμε τα παρακάτω διαγράμματα. Έχουμε 4 διαγράμματα για κάθε λειτουργία και μέγεθος, και το καθένα έχει τις 4 τακτικές με άξονες το Throughput και το μέγεθος των threads.

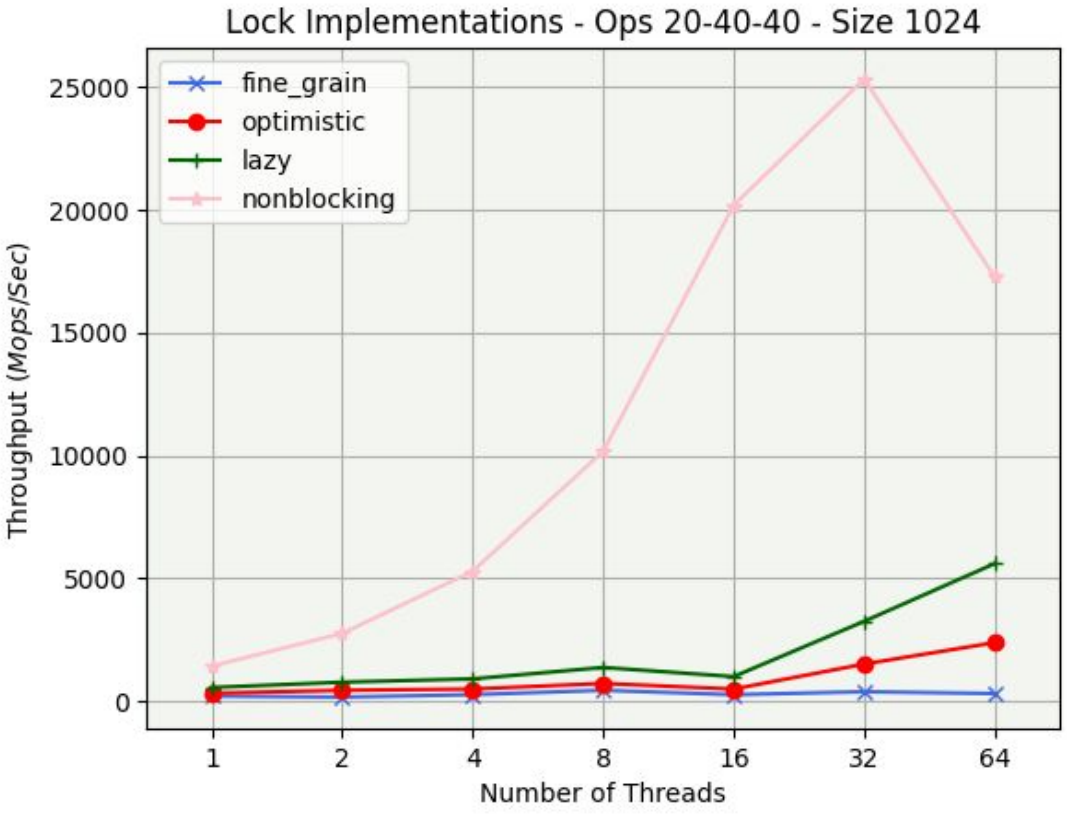
Διαγράμματα Throughput (Operations: 80/10/10, Size = 1024)



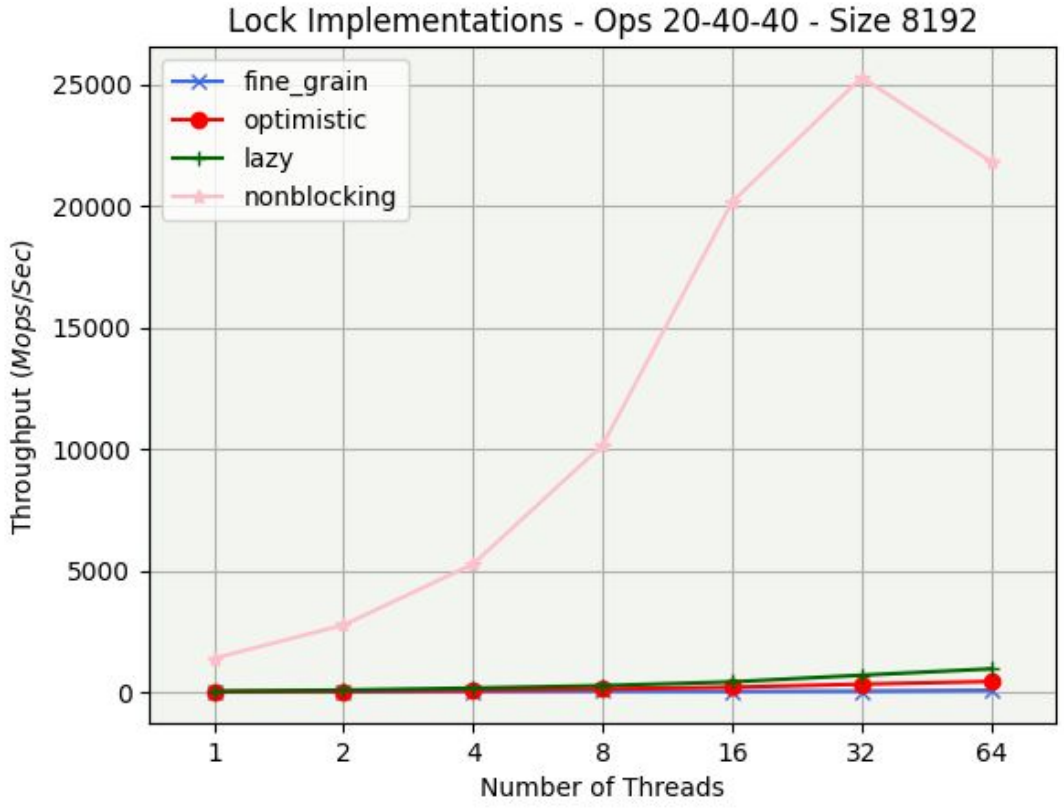
Διαγράμματα Throughput (Operations: 80/10/10, Size = 8192)



Διαγράμματα Throughput (Operations: 20/40/40, Size = 1024)



Διαγράμματα Throughput (Operations: 20/40/40, Size = 8192)



Συμπεράσματα:

Αρχικά όπως βλέπουμε από όλα τα διαγράμματα, η nonblocking υλοποίηση έχει ξεκάθαρα το μεγαλύτερο throughput από όλες τις υπόλοιπες για κάθε μεγεθος threads (εκτός του 1 thread που έχουν παρόμοιο throughput).

Για την nonblocking υλοποίηση, ότι η αλλαγή του μεγέθους της λίστας δεν επηρεάζει την συμπεριφορά και το throughput που επιτυγχάνει. Αυτό που επηρεάζει την συμπεριφορά της όμως είναι το είδος των λειτουργιών.
Για 80% αναζητήσεις και 10% εισαγωγές και διαγραφές κόμβων, έχουμε εκθετική αύξηση του throughput με την αύξηση των threads.
Όμως για 20% αναζητήσεις και 40% εισαγωγές και διαγραφές κόμβων, υπάρχει μια πτώση μετά τα 32 threads του throughput. Επίσης οι τιμές του throughput έχουν πέσει αριθμητικά σχεδόν στο μισό. Αυτό είναι αναμενόμενο, καθώς η διαδικασία της αναζήτησης γίνεται με wait-free τρόπο, οπότε όσο περισσότερες αναζητήσεις έχουμε δεν πρόκειται να περιορίσουν την απόδοση. Όμως όταν έχουμε περισσότερες εγγραφές και διαγραφές τότε όσα περισσότερα threads έχουμε, τόσο περισσότερες οι πιθανότητες να κάνουν συνεχόμενα retries και να δημιουργούνται καθυστερήσεις μειώνοντας έτσι το throughput.

Για τις υπόλοιπες υλοποιήσεις, βλέπουμε ότι έχουν σχεδόν σταθερή απόδοση με μια μικρή αύξηση για πάνω από 16 threads για το μικρό μέγεθος λίστας. Αρχικά βλέπουμε ότι καλύτερη από τις 3 φαίνεται να είναι πάντα η Lazy υλοποίηση, μετά η Optimistic και χειρότερη από όλες η Fine-grain όπως ήταν αναμενόμενο.
Για 80% αναζητήσεις και 10% εισαγωγές και διαγραφές κόμβων, έχουμε σχεδόν σταθερό throughput με μικρή αύξηση όπως αναφέραμε για το μικρό μέγεθος λίστας.
Όμως για 20% αναζητήσεις και 40% εισαγωγές και διαγραφές κόμβων, βλέπουμε ότι όπως και στην nonblocking υλοποίηση έχουμε πτώση των τιμών του throughput με ίδια όμως συμπεριφορά.