Εθνικό Μετσόβιο Πολυτεχνείο

Συστήματα Παραλλήλης Επεξεργασίας

Εργαστηριακή άσκηση 3

Θέματα Συγχρονισμού σε Σύγχρονα Πολυπύρηνα Συστήματα

Θρασύβουλος Ηλιάδης : 03115761

Πέππας Αθανάσιος : 03115749

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 εξετάσετε πώς κάποια χαρακτηριστικά της αρχιτεκτονικής του συστήματος επηρεάζουν την επίδοση των εφαρμογών που εκτελούνται σε αυτά και θα αξιολογήσετε 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.

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

Κάθε νήμα εκτελεί πράξεις πάνω σε διαφορετικά δεδομένα συνεπώς δε χρειάζεται να τα συγχρονίσουμε.

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

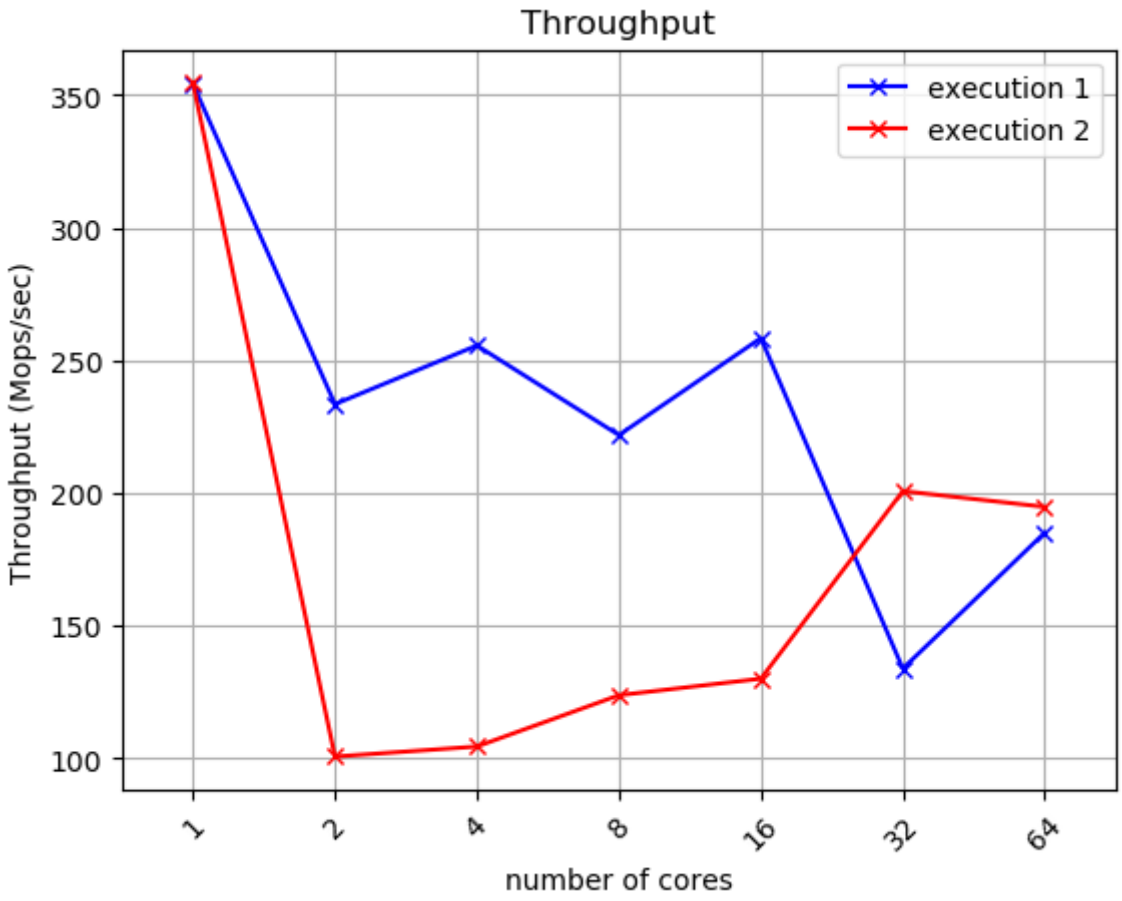
Από τη στιγμή που δεν υπάρχει θέμα συγχρονισμού αναμένουμε η επίδοση να αυξάνει ανάλογα με την αύξηση των νημάτων.

2.3. 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.

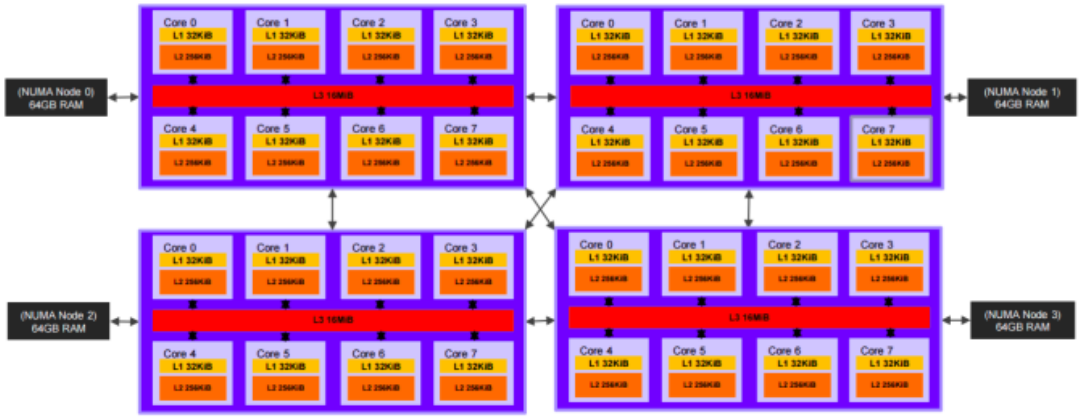
Εκτελούμε το πρόγραμμα `accounts` με βάση τον παρακάτω πίνακα

Αριθμός Νημάτων	Εκτέλεση 1	Εκτέλεση 2
1	0	0
2	0,1	0,8
4	0-3	0,8,16,32
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

Ύστερα από την εκτέλεση του προγράμματος έχουμε τα εξής αποτελέσματα



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.

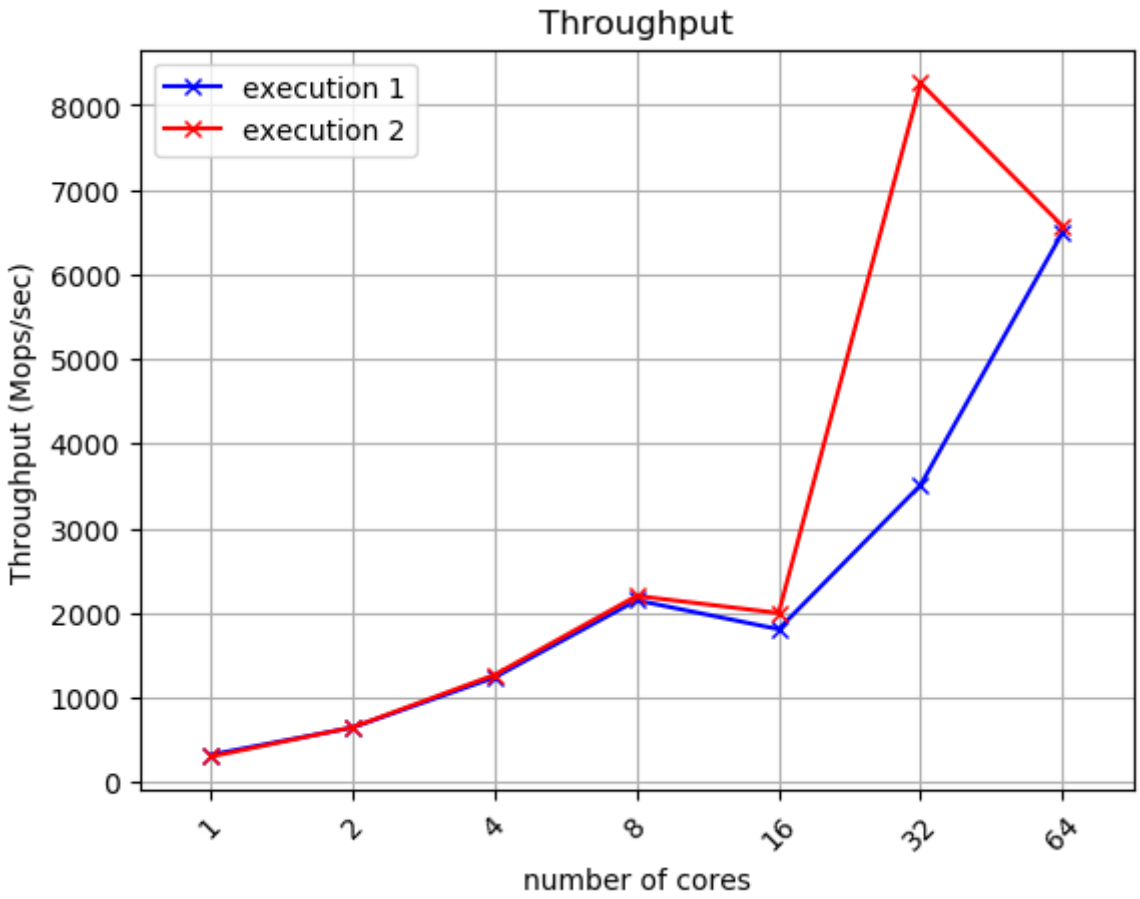
2.4. Η εφαρμογή έχει την συμπεριφορά που αναμένατε; Αν όχι, εξηγήστε γιατί συμβαίνει αυτό και προτείνετε μία λύση. Τροποποιήστε κατάλληλα τον κώδικα και δώστε και πάλι τα αντίστοιχα διαγράμματα για τις δύο εκτελέσεις. Υπόδειξη: πώς αποθηκεύεται ο πίνακας με τους λογαριασμούς στα διάφορα επίπεδα της ιεραρχίας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

Για να μπερέσουμε να αξιοποιήσουμε καλύτερα τις cache μπορούμε να βάλουμε ενα padding στα δεδομένα ώστε κάθε νήμα να εκτελεί σε διαφορετικό block και να μην υπάρχει αληλλεπίδραση μεταξύ των νημάτων.

Παρακάτω φαίνεται ο τροποποιημένος κώδικας

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

Τώρα τα αποτελέσματα που παίρνουμε από τις δύο εκτελέσεις είναι τα εξής



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 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.

3.1. Υλοποιήστε τα ζητούμενα κλειδώματα συμπληρώνοντας τα αντίστοιχα αρχεία της μορφής <lock_ type>_lock.c.

Συγκεκριμένα θα υλοποιήσουμε τα παρακάτω κλειδώματα: (με μαύρο όσα πρεπει να υλοποιήσουμε εμείς)

- *nosync_lock*: Η συγκεκριμένη υλοποίηση δεν παρέχει αμοιβαίο αποκλεισμό και θα χρησιμοποιηθεί ώς άνω όριο για την αξιολόγηση της επίδοσης των υπόλοιπων κλειδωμάτων.

- ***pthread_lock:*** Η συγκεκριμένη υλοποίηση χρησιμοποιεί ένα από τα κλειδώματα που παρέχεται από την βιβλιοθήκη Pthreads (pthread_spinlock_t).

```
#include "lock.h"
#include "/home/parallel/pps/2020-2021/a3/common/alloc.h"
#include <pthread.h>

struct lock_struct {
    pthread_spinlock_t lock;
};

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

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

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

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

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

- *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.

```
#include "lock.h"
#include "/home/parallel/pps/2020-2021/a3/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);
}
```

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

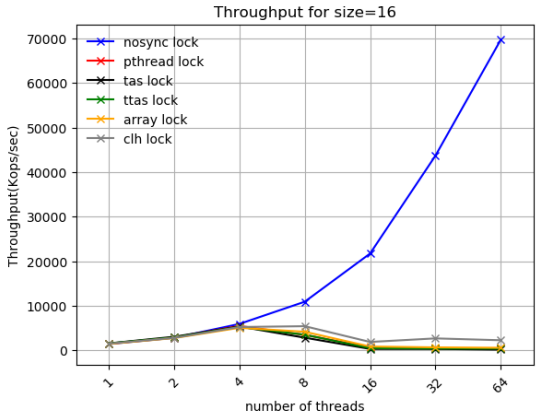
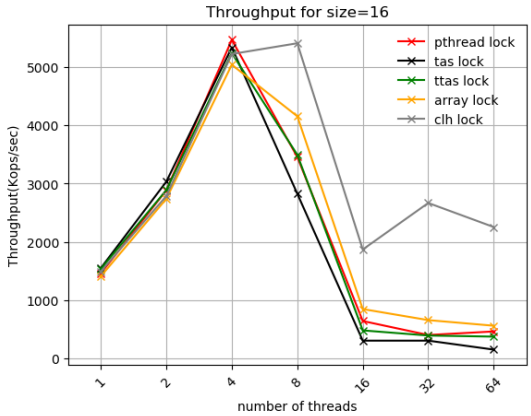
```
#include "lock.h"
#include "/home/parallel/pps/2020-2021/a3/common/alloc.h"
#include <pthread.h>

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;
}
```

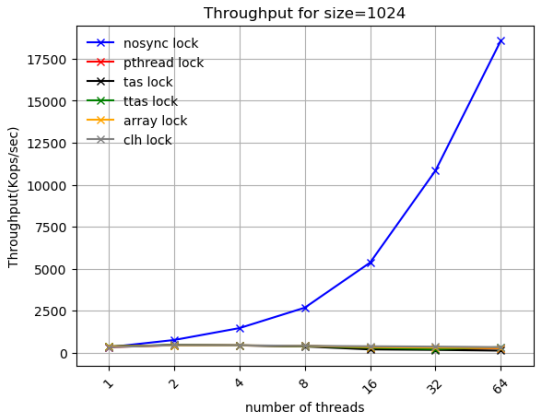
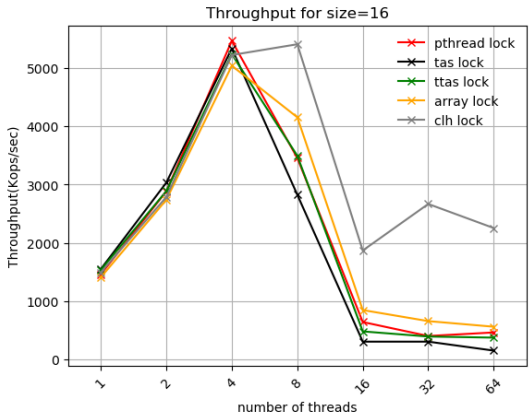
- *clh_lock:* Ένα είδος κλειδώματος που στηρίζεται στη χρήση μίας συνδεδεμένης λίστας.

3.2. 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.

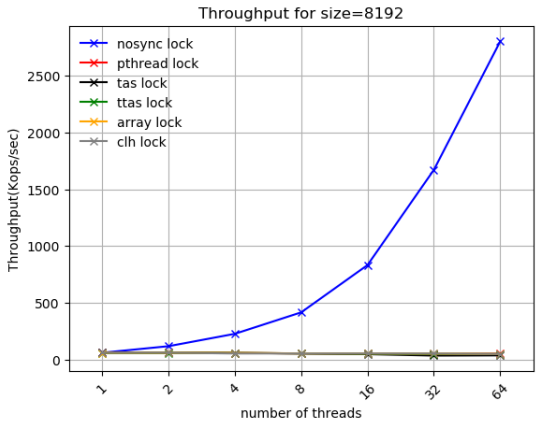
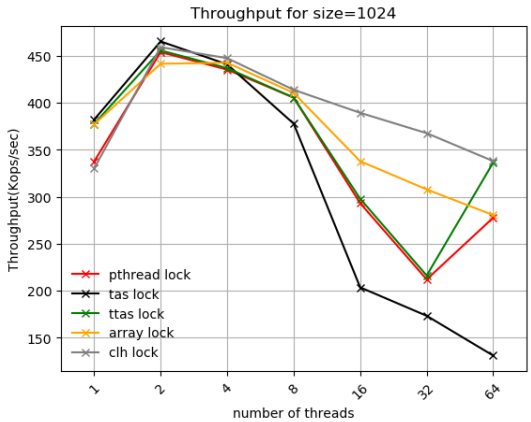
- Λίστα μεγέθους 16

με nosync_lock	χωρίς nosync_lock
	

- Λίστα μεγέθους 1024

με nosync_lock	χωρίς nosync_lock
	

- Λίστα μεγέθους 8192

με nosync_lock	χωρίς nosync_lock
	

Με βάση τα παραπάνω έχουμε τα εξής σχόλια:

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. Επίσης, η συνολική επίδοση μειωνεται όσο το μέγεθος της λίστας αυξάνει καθώς το κρίσιμο κομμάτι παίρνει περισσότερο χρόνο.

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. Τέλος, το ttas lock είναι πάντα αποδοτικότερο από το tas lock επιβεβαιώνοντας την θεωρία.

Συμπερασματικά, μπορούμε να πούμε πως για μικρό αριθμό νημάτων αρκεί να χρησιμοποιήσουμε τα απλά κλειδώματα καθώς δεν υπαχει μεγάλη συμφώρηση και έτσι τα αρνητικά τους δεν εμφανίζονται. Όμως, όταν έχουμε μεγάλο αριθμό νημάτων, η χρήση πιο εξελιγμένων νημάτων που χρησιμοποιούν καποιες δομές δεδομένων, επιτρέπουν την εκτέλεση μεγάλου πλήθους νημάτων με όσο το δυνατόν μικρότερη αλληλεπίδραση μεταξύ τους.

4. Τακτικές συγχρονισμού για δομές δεδομένων

Στο τρίτο και τελευταίο μέρος της άσκησης στόχος είναι η υλοποίηση και η αξιολόγηση των διαφορετικών εναλλακτικών τακτικών για δομές δεδομένων. Συγκεκριμένα έχουμε μια ταξινομημένη συνδεδεμένη λίστα και εκτελούμε σε αυτήν αναζητήσεις, εισαγωγές και διαγραφές.

4.1. Υλοποιήστε τις ζητούμενες λίστες συμπληρώνοντας τα αντίστοιχα αρχεία της μορφής ll<sync type>.c.

Συγκεκριμένα υλοποιούμε τις παρακάτω τακτικές συγχρονισμού.

- *Fine-grain locking*

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

#include "/home/parallel/pps/2020-2021/a3/common/alloc.h"
#include "ll.h"

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

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

/**
 * 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)
{
```

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

    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)
{
    //pthread_spin_lock(&ll->lock);
    int ret = 0;

    ll_node_t *curr, *next;

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

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

int ll_add(ll_t *ll, int key)
```

```
{
    int ret = 0;
    ll_node_t *curr, *next;
    ll_node_t *new_node;

//  if (key < ll->head->key) return 1;

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

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

    if (key != next->key) {
        ret = 1;
        new_node = ll_node_new(key);
        new_node->next = next;
        curr->next = new_node;
    }
    pthread_spin_unlock(&curr->lock);
        pthread_spin_unlock(&next->lock);
    return ret;
}

int ll_remove(ll_t *ll, int key)
{
    int ret=0;
    //pthread_spin_lock(&ll->lock);
        ll_node_t *prev = ll->head;
    pthread_spin_lock(&prev->lock);

        ll_node_t *curr = prev->next;
        pthread_spin_lock(&curr->lock);
        //pthread_spin_unlock(&ll->lock);

    while (curr->key < key && curr->next != NULL){
        pthread_spin_unlock(&prev->lock);
            prev = curr;
            curr = curr->next;
            pthread_spin_lock(&curr->lock);
    }
    if (key == curr->key){
            prev->next = curr->next;
        pthread_spin_unlock(&prev->lock);
        pthread_spin_unlock(&curr->lock);
        ll_node_free(curr);
        ret = 1;
        }
    else{
        pthread_spin_unlock(&prev->lock);
```

```
        pthread_spin_unlock(&curr->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 "/home/parallel/pps/2020-2021/a3/common/alloc.h"
#include "ll.h"

typedef struct ll_node {
    int key;
    struct ll_node *next;
    pthread_spinlock_t lock;
    /* other fields here? */
} 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)
{
    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 validate(ll_t *ll, ll_node_t *pred, ll_node_t *curr){
        ll_node_t *node = ll->head;

        while (node->key <= pred->key && node != NULL) {
                if (node == pred) return (pred->next==curr);
                node = node->next;
        }
        return 0;
}

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

    while(1) {
                pred = ll->head;
```

```
                curr = pred->next;

                while (curr != NULL && curr->key <= key) {
            if (curr->key == key)
                break;
                        pred = curr;
                        curr = curr->next;
                }

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

                if (validate(ll, pred, curr)){
            pthread_spin_unlock(&(curr->lock));
                    pthread_spin_unlock(&(pred->lock));
            return (curr->key == key);
        }

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

int ll_add(ll_t *ll, int key)
{
        ll_node_t *pred, *curr, *new_node;

        while (1){
                pred = ll->head;
                curr = pred->next;
                while (curr->key <= key && curr!=NULL){
                        //if ( key == curr->key) break;
                        pred = curr;
                        curr = curr->next;
                }

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

                if (validate(ll,pred,curr)){
                        if (curr->key != key){
                                new_node = ll_node_new(key);
                pred->next = new_node;
                new_node->next = curr;
                pthread_spin_unlock(&curr->lock);
                pthread_spin_unlock(&pred->lock);
                                return 1;
                        }
                        else {
                pthread_spin_unlock(&curr->lock);
                pthread_spin_unlock(&pred->lock);
                                return 0;
                        }
                }
                pthread_spin_unlock(&pred->lock);
                pthread_spin_unlock(&curr->lock);
        }
}
```

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

    while (1){
        pred = ll->head;
        curr = pred->next;
        while (curr->key <=key && curr!=NULL){
            if ( key == curr->key) break;
            pred = curr;
            curr = curr->next;
        }

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

        if (validate(ll,pred,curr)){
            if (curr->key == key){
                pred->next = curr->next;
                pthread_spin_unlock(&curr->lock);
                pthread_spin_unlock(&pred->lock);
                return 1;
            }
            else {
                pthread_spin_unlock(&curr->lock);
                pthread_spin_unlock(&pred->lock);
                return 0;
            }
        }
        pthread_spin_unlock(&pred->lock);
        pthread_spin_unlock(&curr->lock);
    }
}

/**
 * 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>
```

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

#include "/home/parallel/pps/2020-2021/a3/common/alloc.h"
#include "ll.h"

typedef struct ll_node {
    int key;
    struct ll_node *next;
    pthread_spinlock_t lock;
    int marked; // 0 --> init // 1 --> deleted
} 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;
    ret->marked = 0;
    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;
}
```

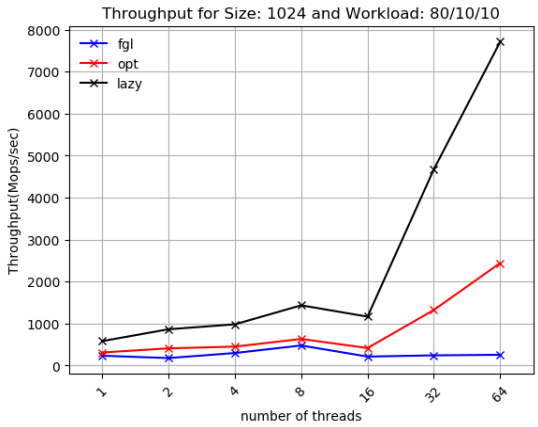
```c
/**
 * Free a linked list and all its contained nodes.
 **/
void ll_free(ll_t *ll)
{
    ll_node_t *next, *curr = ll->head;
    while (curr) {
        next = curr->next;
        ll_node_free(curr);
        curr = next;
    }
    XFREE(ll);
}

int validate(ll_node_t *pred, ll_node_t *curr){
    return (!pred->marked && !curr->marked && pred->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->key==key && !curr->marked);
}

int ll_add(ll_t *ll, int key)
{
    ll_node_t *pred, *curr, *new_node;

    while (1){
        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 (validate(pred, curr)){
            if (curr->key != key){
                new_node = ll_node_new(key);
                pred->next = new_node;
                new_node->next = curr;
                pthread_spin_unlock(&pred->lock);
                pthread_spin_unlock(&curr->lock);
                return 1;
            }
            else {
                pthread_spin_unlock(&pred->lock);
                pthread_spin_unlock(&curr->lock);
                return 0;
```

```
            }
        }
        pthread_spin_unlock(&pred->lock);
        pthread_spin_unlock(&curr->lock);
    }
    return 0;
}

int ll_remove(ll_t *ll, int key)
{
    ll_node_t *pred, *curr;

    while (1){
        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 (validate(pred,curr)){
            if (curr->key == key){
                curr->marked = 1;
                pred->next = curr->next;
                pthread_spin_unlock(&pred->lock);
                pthread_spin_unlock(&curr->lock);
                return 1;
            }
            else{
                pthread_spin_unlock(&pred->lock);
                pthread_spin_unlock(&curr->lock);
                return 0;
            }
        }
        pthread_spin_unlock(&pred->lock);
        pthread_spin_unlock(&curr->lock);
    }
}

/**
 * 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. 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 (αναζητήσεις-εισαγωγές-διαγραφές) 80-10-10 και 20-40-40. Παρουσιάστε τα αποτελέσματά σας σε διαγράμματα και εξηγήστε την συμπεριφορά της εφαρμογής για κάθε κλείδωμα.

Παρακάτω βλέπουμε τα αποτελέσματα από τις εκτελέσεις

- *Συνδυασμός λειτουργιών 80/10/10*

1024x1024	8192x8192
	

- *Συνδυασμός λειτουργιών 20/40/40*

1024x1024	8192x8192
	

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). Όσο αυξάνονται τα νήματα, η διαφορά τους μεγαλώνει καθώς έχουμε όλο και περισσότερες λειτουργίες και όλο και περισσότερες καταστάσεις συναγωνισμού. Ως προς το μέγεθος της λίστας, προφανώς όταν μεγαλώνει το μέγεθός της, το throughput πέφτει καθώς έχουμε να διατρέξουμε μία μεγαλύτερη λίστα. 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.