Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Συστήματα Παράλληλης Επεξεργασίας

5η Εργαστηριακή Άσκηση

9ο Εξάμηνο - Ακαδημαϊκό Έτος 2021-2022

Παγώνης Γεώργιος - 03117030

Γιαννόπουλος Εμμανουήλ - 03117031

28 Φεβρουαρίου 2022

1 Λογαριασμοί Τράπεζας

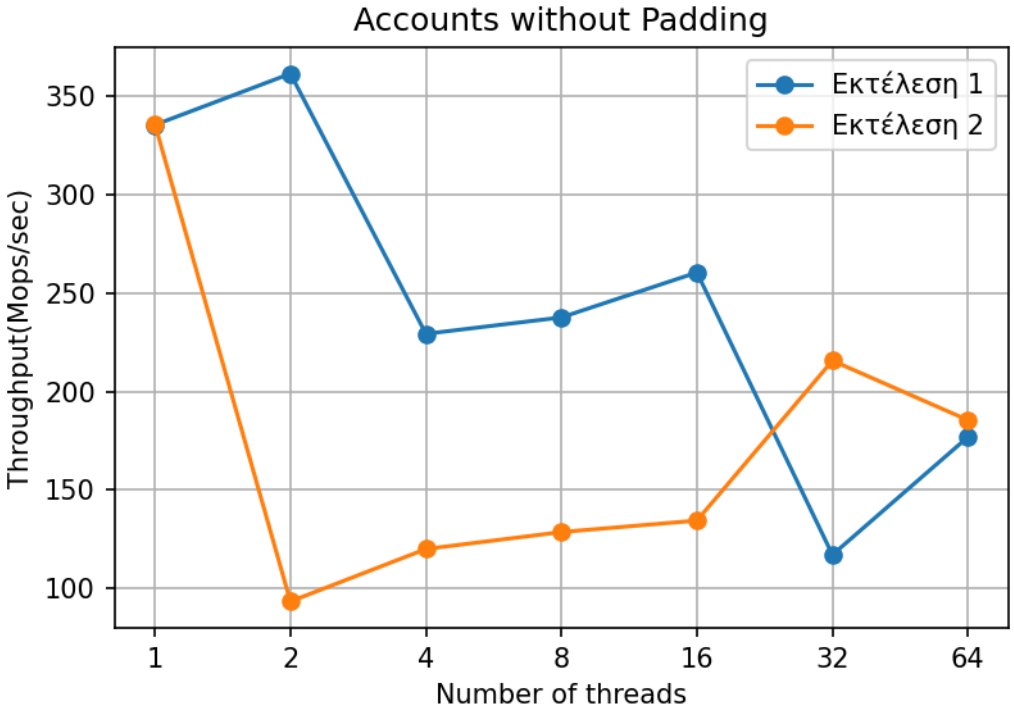
1. Υπάρχει ανάγκη για συγχρονισμό ανάμεσα στα νήματα της εφαρμογής;

   Όχι, εφόσον το κάθε νήμα εργάζεται σε διαφορετικό λογαριασμό.

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

   Θα περιμέναμε το speedup να αυξάνεται γραμμικά ως προς τα νήματα εφόσον δε χρησιμοποιούμε κάποιο σχήμα συγχρονισμού.

1.1 Αρχική εκτέλεση



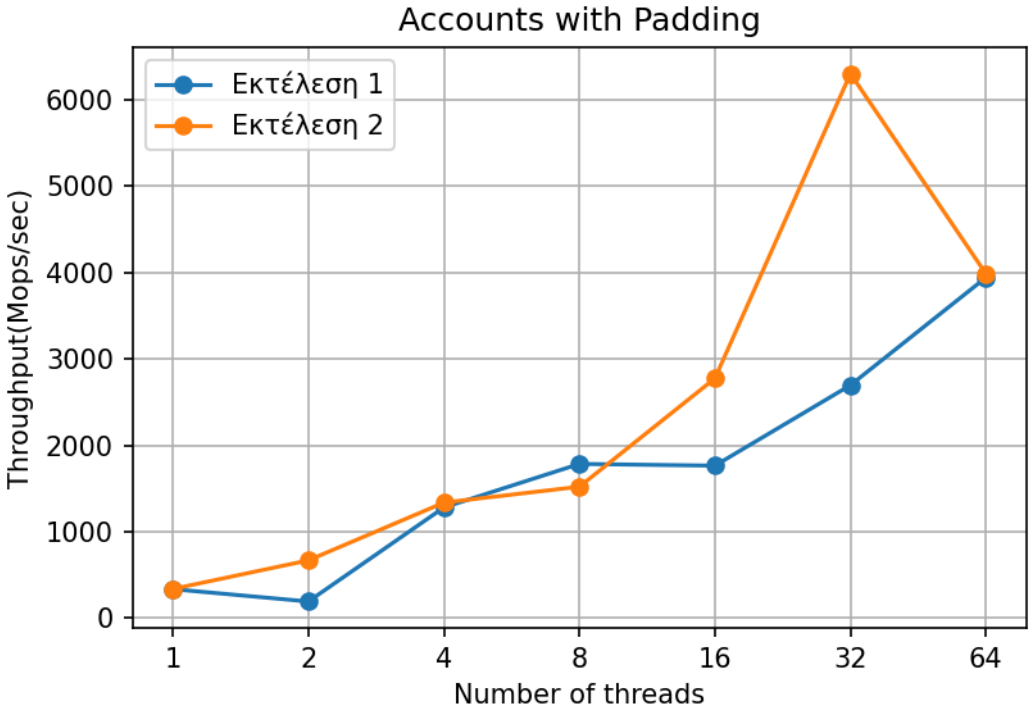
Παρατηρούμε ότι οι εκτελέσεις διαφέρουν ως προς το throughput λόγω του διαφορετικού thread affinity που ορίζεται σε κάθε εκτέλεση. Συγκεκριμένα, γνωρίζουμε από την αρχιτεκτονική του sandman ότι τα νήματα που τρέχουν στον ίδιο πυρήνα μοιράζονται όλα τα στάδια της ιεραρχίας κρυφής μνήμης, ενώ τα νήματα που τρέχουν στο ίδιο socket μοιράζονται την L3 cache. Λόγω του παραπάνω, είναι λογικό η εκτέλεση 1 που χρησιμοποιεί κοντινότερους πυρήνες να είναι καλύτερη από την εκτέλεση 2.

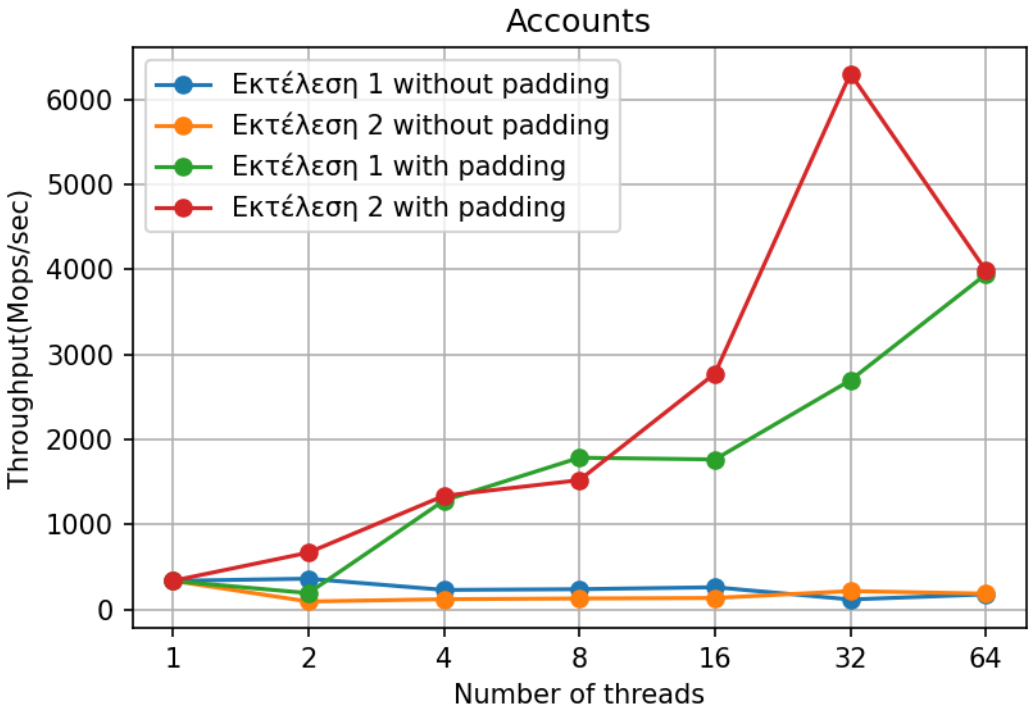
1.2 Βελτιωμένη εκτέλεση

Παραμένει όμως το ερώτημα: Γιατί το πρόγραμμα δεν κλιμακώνει όσο προσθέτουμε νήματα (αντιθέτως - γίνεται χειρότερο). Το πρόβλημα οφείλεται στο ότι κάθε νήμα εκτελεί πράξεις πάνω σε 4 bytes. Το μέγεθος αυτό είναι μικρότερο από το block size οποιουδήποτε επιπέδου στην ιεραρχία της κρυφής μνήμης. Αυτό σημαίνει ότι το κάθε νήμα φέρνει στην κρυφή μνήμη του και κάποιες γειτονικές θέσεις μνήμης. Έστω νήμα A που έχει φέρει στην κρυφή του μνήμη ένα block με το δικό του λογαριασμό, το λογαριασμό του νήματος B και κάποιους άλλους. Όταν νήμα B γράψει στη γειτονική θέση μνήμης,

θα τρέξει ένα cache coherence protocol σύμφωνα με το οποίο το νήμα A θα πρέπει να κάνει update την cache του. Εκεί χάνεται ο χρόνος και έχουμε χαμηλότερο throughput.

Λύση: Padding! Γεμίζουμε θέσεις του πίνακα με την τιμή 0 και "σπάμε" τους λογαριασμούς έτσι ώστε κάθε νήμα να φέρνει στην cache μόνο τον δικό του λογαριασμό και ghost cells.





Πράγματι, τώρα βλέπουμε ότι η εφαρμογή μας κλιμακώνει πολύ καλύτερα. Συγκρίνοντας δε την επίδοση πριν και μετά το padding, βλέπουμε πόσο μεγάλη διαφορά προκαλεί η λύση μας. Προφανώς για να λύσουμε το πρόβλημα χρησιμοποιήσαμε περισσότερη μνήμη, οπότε η λύση που προτείναμε έρχεται με ένα trade-off.

1.3 Κώδικας

Ο παρακάτω κώδικας προστέθηκε για το padding.

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

2 Αμοιβαίος Αποκλεισμός - Κλειδώματα

2.1 Κώδικας

2.1.1 Array lock

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

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

__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;
    lock->flag[0] = true;
    for(i=1;i<nthreads;i++) lock->flag[i] = false;
    lock->tail = 0;
```

```
    return lock;
}

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

void lock_acquire(lock_t *lock)
{
    // take the tail value (%lock->size to be circular)
    // and increament the tail value atomically
    int slot = __sync_fetch_and_add(&(lock->tail),1) %lock->size;
    mySlotIndex = slot; // Save my slot to thread value;
    while (!lock->flag[slot]){};
    // First how would come will take mySlotIndex = 0 = tail , would do tail++
    // check if lock->flag[mySlotIndex] = true
    // The second will take mySlotIndex=tail=1, would do tail++
    // check if lock->flag[mySlotIndex]=true
}

void lock_release(lock_t *lock)
{
    //Get my slot
    int slot = mySlotIndex;
    //Make my slot false
    lock->flag[slot] = false;
    // Make the next
    lock->flag[(slot+1)%lock->size] = true;
}
```

2.1.2 Pthread lock

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

struct lock_struct {
    pthread_spinlock_t spin;
};

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

    XMALLOC(lock, 1);
    int pshared=0;
```

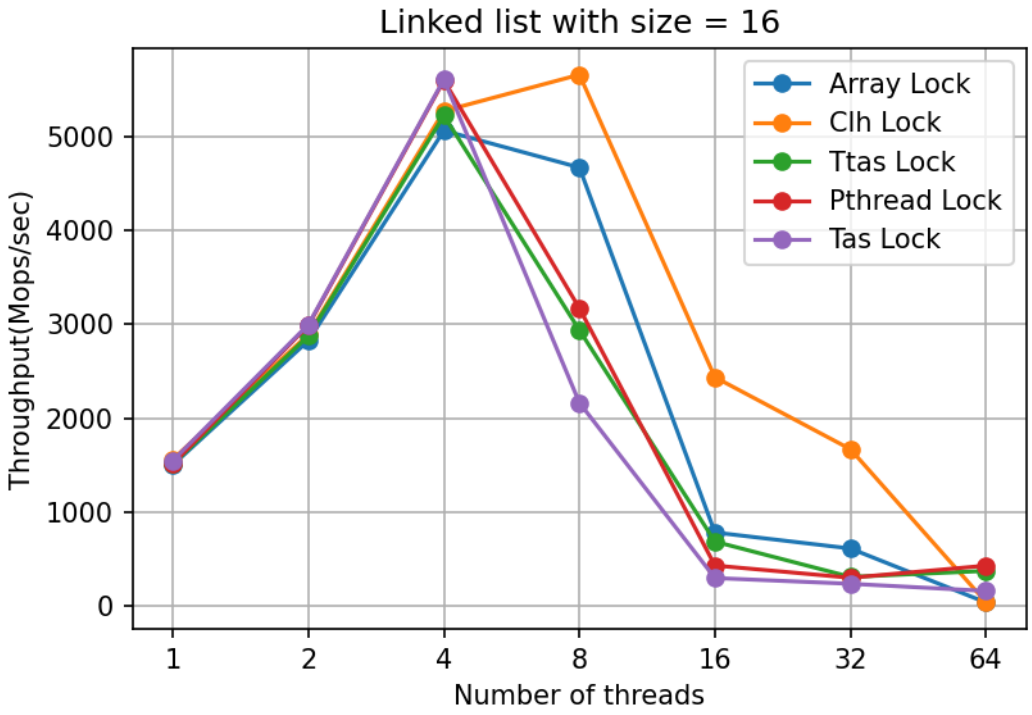
```
    /* other initializations here. */
    if(pthread_spin_init(&lock->spin,pshared)!=0){
        printf("Couldn't init the spinlock\n");
    }
    return lock;
}

void lock_free(lock_t *lock)
{
    if(pthread_spin_destroy(&lock->spin)!=0){
        printf("Couldn't destroy the spinlock\n");
    }
    XFREE(lock);
}

void lock_acquire(lock_t *lock)
{
    if(pthread_spin_lock(&lock->spin)!=0){
        printf("Couldn't take the lock\n");
    }
}

void lock_release(lock_t *lock)
{
    if(pthread_spin_unlock(&lock->spin)!=0){
        printf("Couldn't unlock the lock");
    }
}
```

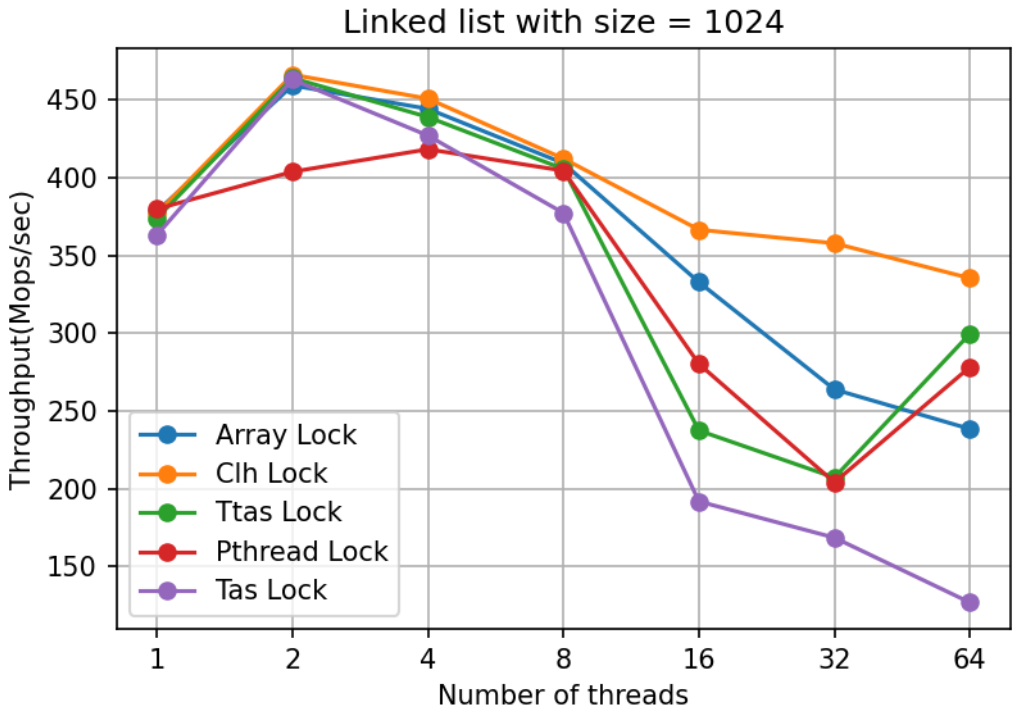
2.1.3 Ttas lock

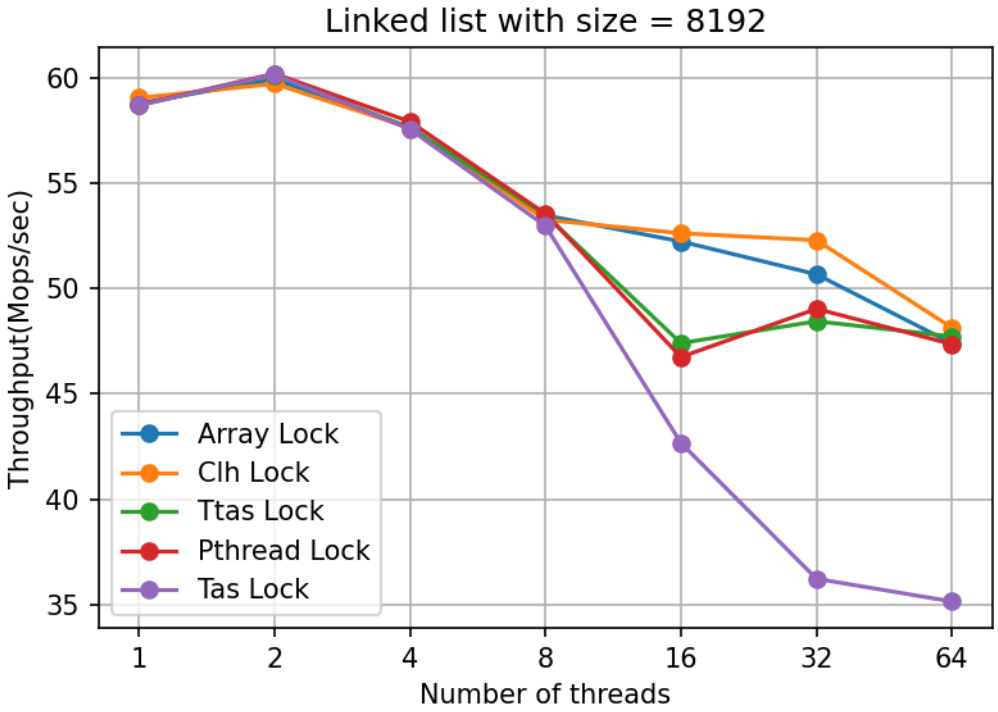
```
#include "../common/alloc.h"
#include "lock.h"
#include <stdbool.h>

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;
    bool sit = true;
    while(sit){
        // Check constantly if the l->state become UNLOCKED

        while(l->state==LOCKED){};

        // This change the value of l->state to LOCKED and return the previous
   value
        // if l->state was UNLOCKED then the process can have the lock
        // if not the process couldn't make it in time because another process
        // reached first so this one must wait again

        if (__sync_lock_test_and_set(&l->state,LOCKED)==UNLOCKED) sit = false;
    }

}

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

}
```

2.2 Διαγράμματα







Παρατηρήσεις

Nosync lock: Το nosync όπως περιμένουμε έχει το μεγαλύτερο throughput από όλα αφού στην ουσία δεν πραγματοποιεί συγχρονισμό. Βάζοντάς το στα διαγράμματα είχε τόσο υψηλότερο throughput που τα υπόλοιπα δεν φαίνονταν καλά σε κλίμακα. Επομένως για λόγους καλύτερης εποπτείας παραλείπεται από τα διαγράμματα.

Για μικρό αριθμό νημάτων, τα διαφορετικά κλειδώματα δεν εμφανίζουν μεγάλες αποκλίσεις στην επίδοσή τους, καθώς δεν υπάρχει αρκετή αναμονή για να εισέλθουν στο κρίσιμο τμήμα ώστε να παίζουν μεγάλο ρόλο οι διαφορές στην υλοποίηση. Για μεγαλύτερα νήματα όμως, βλέπουμε ότι:

- Τα καλύτερα locks είναι τα array και clh locks, ακολουθούμενα από το pthread και το ttas lock. Αυτό συμβαίνει γιατί τα 2 πρώτα χρησιμοποιούν συγκεκριμένες δομές δεδομένων για να επιταχύνουν την περίοδο αναμονής των νημάτων στο κλείδωμα.
- Βλέπουμε ότι το tas lock είναι σε κάθε περίπτωση πιο αργό από το ttas lock, πράγμα που αναμέναμε αφού το δεύτερο δεν καταναλώνει πολλούς πόρους από το bus τεστάροντας συνέχεια αν είναι έτοιμο το κλείδωμα, όπως κάνει το tas.
- Παρατηρούμε επίσης ότι στη μεγαλύτερη λίστα οι αποκλίσεις γίνονται μικρότερες, πιθανότατα λόγω του μεγάλου χρονικού διαστήματος που χρειάζεται για να φτάσουμε στο σωστό σημείο της λίστας.

3 Τακτικές συγχρονισμού για δομές δεδομένων

3.1 Κώδικας

3.1.1 Fine-grain locking

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

  curr = ll->head;
    pthread_spin_lock(&curr->spin);
  next= curr->next;
    pthread_spin_lock(&next->spin);
    int ret = 0;

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

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

    pthread_spin_unlock(&curr->spin);
    pthread_spin_unlock(&next->spin);

    return ret;
}

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

    ll_node_t *curr, *next;
    ll_node_t *new_node;

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

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

    if (key != next->key) {
        ret = 1;
        new_node = ll_node_new(key);
        new_node->next = next;
```

```
        curr->next = new_node;
    }
    pthread_spin_unlock(&curr->spin);
    pthread_spin_unlock(&next->spin);

    return ret;
}

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

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

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

    if (key == next->key) {
        ret = 1;
        curr->next = next->next;
        ll_node_free(next);
    }
    else {
        pthread_spin_unlock(&next->spin);
    }
    pthread_spin_unlock(&curr->spin);


    return ret;
}
```

3.1.2 Optimistic synchronization

```
bool validate(ll_t *ll,ll_node_t *pred,ll_node_t *curr){
    ll_node_t *node = ll->head;

    while(node->key <= pred->key){
        if (node == pred)
            return (pred->next == curr);
        node = node->next;
```

```
    }
    return 0;
}


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

    while(true){
        curr = ll->head;
        next = curr->next;
        while (next->key < key){
            curr = next;
            next = curr->next;
        }

        pthread_spin_lock(&curr->spin);
        pthread_spin_lock(&next->spin);

        if(validate(ll,curr,next)){
            int ret = (curr->key ==key);
            pthread_spin_unlock(&curr->spin);
            pthread_spin_unlock(&next->spin);
            return ret;
        }
        pthread_spin_unlock(&curr->spin);
        pthread_spin_unlock(&next->spin);
    }
    return 0;
}

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

    while(true){
        curr = ll->head;
        next = curr->next;
        while (next->key < key){
            curr = next;
            next = curr->next;
        }
```

```
        pthread_spin_lock(&curr->spin);
        pthread_spin_lock(&next->spin);

        if(validate(ll,curr,next)){
            if(key != next->key){
                ret = 1;
                new_node = ll_node_new(key);
                new_node->next = next;
                curr->next = new_node;
                pthread_spin_unlock(&curr->spin);
                pthread_spin_unlock(&next->spin);
                return ret;

            }
            pthread_spin_unlock(&curr->spin);
            pthread_spin_unlock(&next->spin);
            return ret;
        }
        pthread_spin_unlock(&curr->spin);
        pthread_spin_unlock(&next->spin);
    }

    return 0;
}

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

    while(true){
        curr = ll->head;
        next = curr->next;
        while (next->key < key){
            curr = next;
            next = curr->next;
        }

        pthread_spin_lock(&curr->spin);
        pthread_spin_lock(&next->spin);

        if(validate(ll,curr,next)){
            if(key == next->key){
                ret = 1;
                curr->next = next->next;
                pthread_spin_unlock(&next->spin);
                pthread_spin_unlock(&curr->spin);
```

```
                return ret;

            }
            else {
                pthread_spin_unlock(&curr->spin);
                pthread_spin_unlock(&next->spin);
                return ret;
            }
        }
        pthread_spin_unlock(&curr->spin);
        pthread_spin_unlock(&next->spin);
    }

    return 0;
}
```

3.1.3 Lazy synchronization

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

    while(true){
        curr = ll->head;
        next = curr->next;
        while (next->key < key){
            curr = next;
            next = curr->next;
        }

        pthread_spin_lock(&curr->spin);
```

```
        pthread_spin_lock(&next->spin);

        if(validate(curr,next)){
            if(key != next->key){
                ret = 1;
                new_node = ll_node_new(key);
                new_node->next = next;
                curr->next = new_node;
                pthread_spin_unlock(&curr->spin);
                pthread_spin_unlock(&next->spin);
                return ret;

            }
            pthread_spin_unlock(&curr->spin);
            pthread_spin_unlock(&next->spin);
            return ret;
        }
        pthread_spin_unlock(&curr->spin);
        pthread_spin_unlock(&next->spin);
    }

    return 0;
}

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

    while(true){
        curr = ll->head;
        next = curr->next;
        while (next->key < key){
            curr = next;
            next = curr->next;
        }

        pthread_spin_lock(&curr->spin);
        pthread_spin_lock(&next->spin);

        if(validate(curr,next)){
            if(key == next->key){
                ret = 1;
                next->marked = true;
                curr->next = next->next;
                pthread_spin_unlock(&next->spin);
                pthread_spin_unlock(&curr->spin);
```

```
                return ret;

            }
            else {
                pthread_spin_unlock(&curr->spin);
                pthread_spin_unlock(&next->spin);
                return ret;
            }
        }
        pthread_spin_unlock(&curr->spin);
        pthread_spin_unlock(&next->spin);
    }

    return 0;
}
```

3.1.4 Non-blocking synchronization

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

#include "../common/alloc.h"
#include "ll.h"
#define TRUE 1
#define FALSE 0
#define OR_TRUE(p) ((ll_node_t *) ((uintptr_t) (p) | (uintptr_t) TRUE)) //
   pointer with last bit 1
#define OR_FALSE(p) ((ll_node_t *) ((uintptr_t) (p) | (uintptr_t) FALSE))   //
   pointer with last bit 0
#define AND_TRUE(p) ((ll_node_t *) ((uintptr_t) (p) & (uintptr_t) TRUE))    //
   return all with mark
#define AND_FALSE(p) ((ll_node_t *) ((uintptr_t) (p) & (uintptr_t) FALSE))  //
   return pointer with mark 0
#define LAST_ZERO(p) ((ll_node_t *) ((uintptr_t) (p) & (uintptr_t) ~TRUE))  //
   remove pointer bits return last bit

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

typedef struct window {
    ll_node_t *pred,*curr;
} window;
```

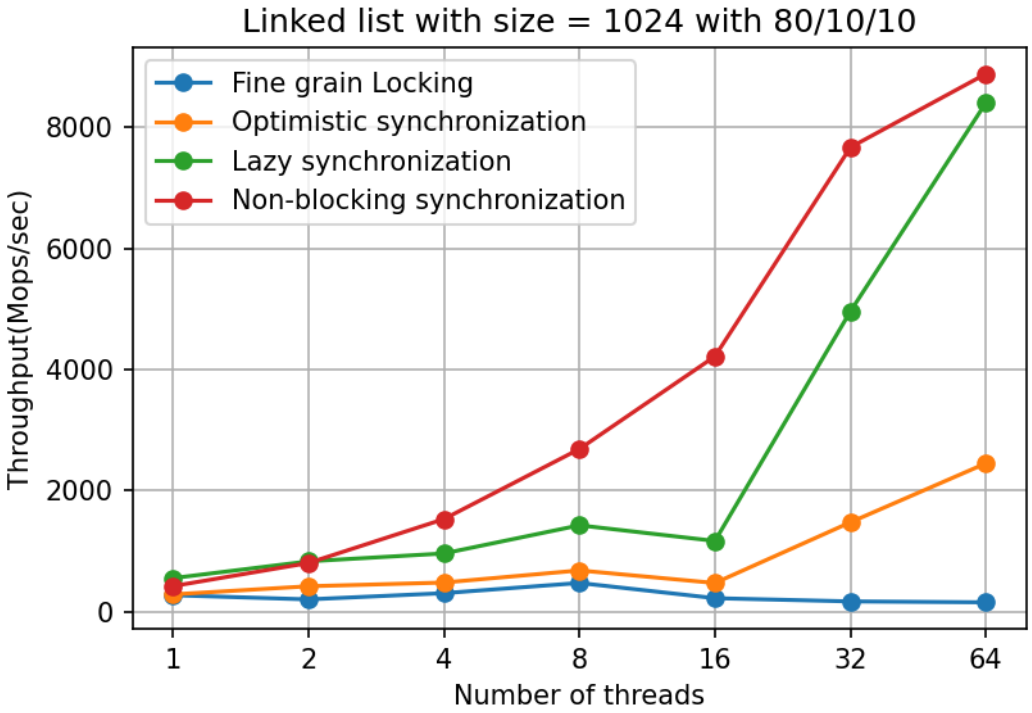
```c
struct linked_list {
    ll_node_t *head;
    /* other fields here? */
};

/**
 * 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? */

    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)
```

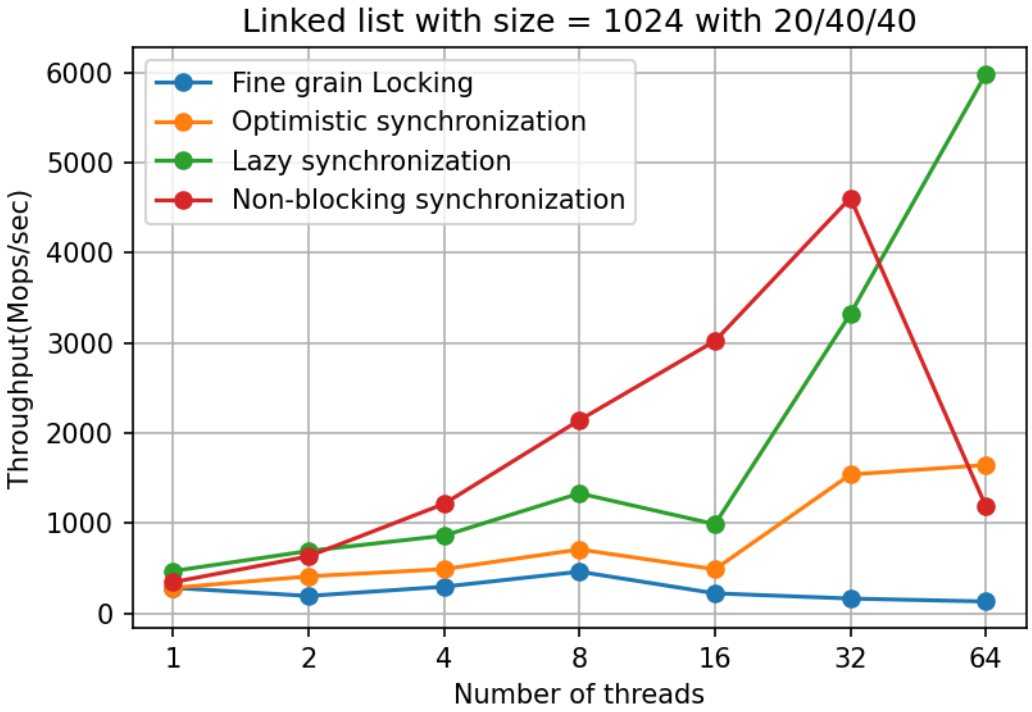
```
{
    ll_node_t *next, *curr = ll->head;
    while (curr) {
        next = curr->next;
        ll_node_free(curr);
        curr = next;
    }
    XFREE(ll);
}

window* find (window* win,ll_node_t* head,int key)
{
    ll_node_t *pred,*curr,*succ;
    pred=NULL;
    curr=NULL;
    succ=NULL;
    int marked[]={0};   //marked = false
    int snip;
    retry: while (1) {
        pred = head;
        curr = LAST_ZERO(pred->next);
        while (1) {
            marked[0] = AND_TRUE(curr->next);               //atomic get
            succ = LAST_ZERO(curr->next);
            while (marked[0]) { //fysikh diagrafh komvwn
                snip = __sync_bool_compare_and_swap(&(pred->next),LAST_ZERO(pred
   ->next),LAST_ZERO(pred->next))
                            && __sync_bool_compare_and_swap(&(pred->next), curr,
    succ);
                if (!snip) goto retry;
                curr = succ;
                marked[0] = AND_TRUE(curr->next);
                succ = LAST_ZERO(curr->next);
                //ll_node_free(curr);
            }
            if (curr->key >= key) {
                win->pred = pred;
                win->curr = curr;
                return win;
            }
            pred = curr;
            curr = succ;
        }
    }
}

int ll_contains(ll_t *ll, int key)
{
```

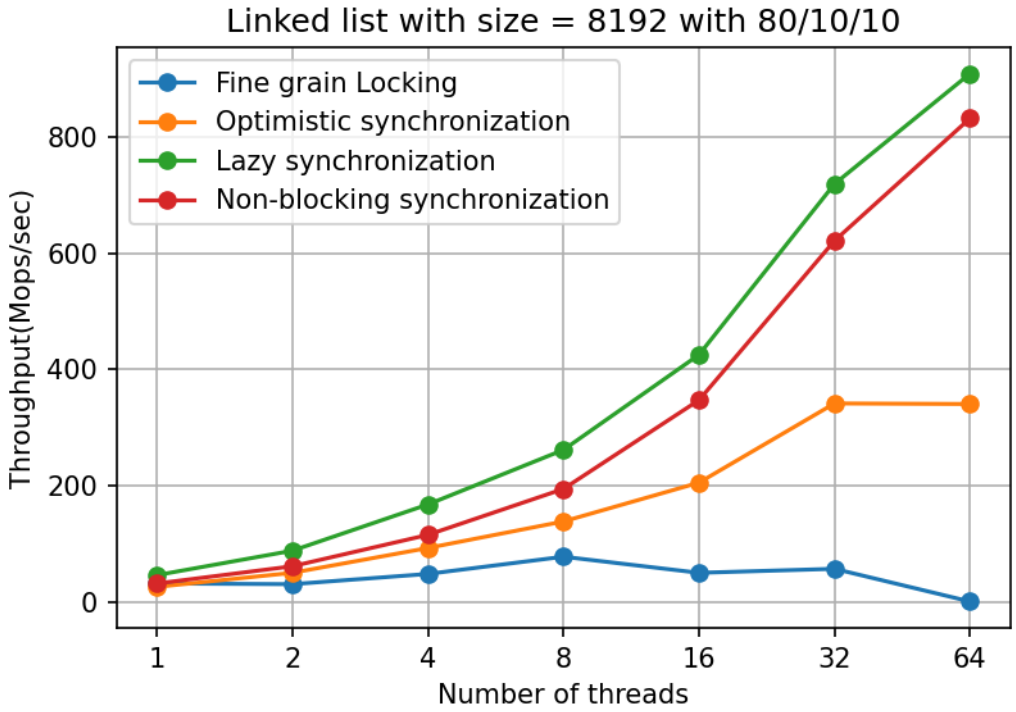
```
    int marked[]={0};
    ll_node_t *curr;
    curr =  ll->head;
    while (curr->key <key){
        curr = LAST_ZERO(curr->next);
    }
    marked[0]=AND_TRUE(curr->next);
    return (curr->key == key && !marked[0]);
}

int ll_add(ll_t *ll, int key)
{
    int splice;
    window* win=(window*)malloc(sizeof(window));
    ll_node_t *pred,*curr,*new_node;
    while (1) {
        win = find(win,ll->head,key);
        pred = win->pred;
        curr = win->curr;
        if (curr->key == key) {
            return 0;
        }
        else {
            new_node=ll_node_new(key);
            new_node->next = curr;
            if (__sync_bool_compare_and_swap(&pred->next, LAST_ZERO(pred->next),
    LAST_ZERO(pred->next))
                && __sync_bool_compare_and_swap(&pred->next, curr, new_node))
   return 1;
        }
    }
}

int ll_remove(ll_t *ll, int key)
{
    int snip;
    window *win=(window*)malloc(sizeof(window));
    while (1) {
        win = find(win,ll->head,key);
        ll_node_t *pred,*curr,*succ;
        pred = win->pred;
        curr = win->curr;
        if (curr->key != key) {
            //free(win);
            return 0;
        }
        else {
            succ = LAST_ZERO(curr->next);
```

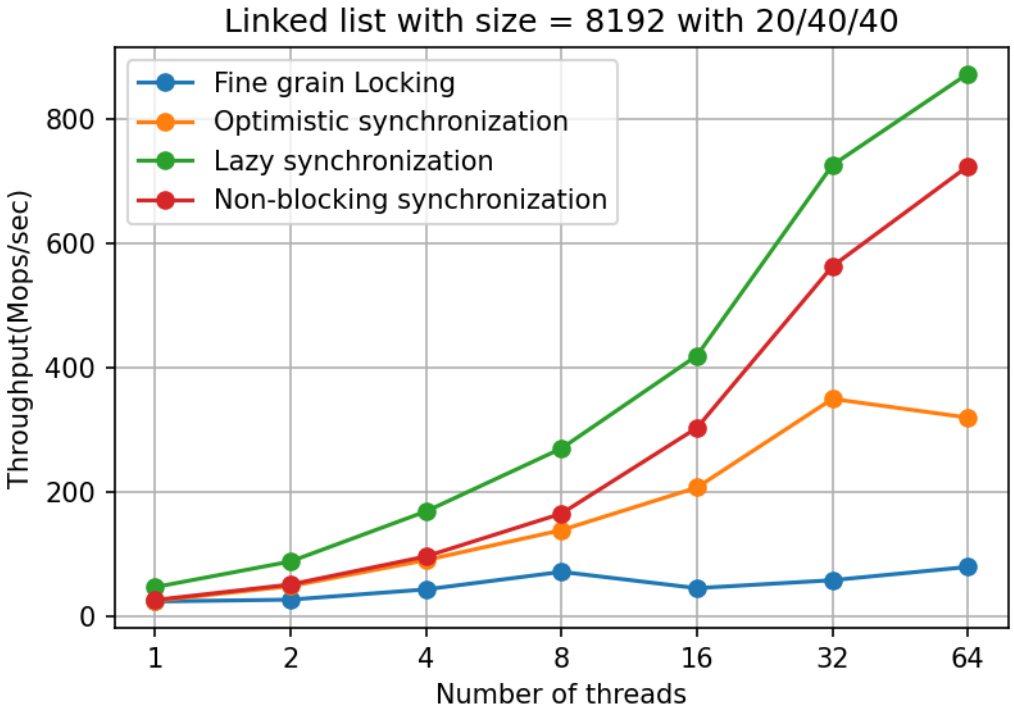
```
            snip = __sync_bool_compare_and_swap(&(curr->next),LAST_ZERO(succ),
    OR_TRUE(succ));
            if (!snip) continue;
            __sync_bool_compare_and_swap(&pred->next, LAST_ZERO(pred->next),
    LAST_ZERO(pred->next))
                && __sync_bool_compare_and_swap(&pred->next, curr, succ);
            //ll_node_free(curr);
            //free(win);
            return 1;
        }
    }
}
```

3.2 Διαγράμματα




Linked list with size = 1024 with 20/40/40
Fine grain Locking
Optimistic synchronization
Lazy synchronization
Non-blocking synchronization
Throughput(Mops/sec)
0
1000
2000
3000
4000
5000
6000
1
2
4
8
16
32
64
Number of threads


Linked list with size = 8192 with 80/10/10
Fine grain Locking
Optimistic synchronization
Lazy synchronization
Non-blocking synchronization
Throughput(Mops/sec)
0
200
400
600
800
1
2
4
8
16
32
64
Number of threads



Παρατηρήσεις

- Τα πιο αποδοτικά σχήματα συγχρονισμού είναι το Lazy synchronization και το Non-blocking synchronization. Η επίδοσή τους εναλλάσσεται από τη μικρή στη μεγάλη λίστα και πιθανότατα αυτό οφείλεται στον αριθμό των retries που χρειάζεται να κάνει το σχήμα non-blocking.
- Το fine-grain locking είναι πολύ χειρότερο σε throughput από τα άλλα 3 και μάλιστα ενώ κλιμακώνει μέχρι το 8 μετά η επίδοσή του πέφτει. Αυτό οφείλεται στο hand-over-hand locking και τα πολλαπλά κλειδώματα που καθυστερούν το πρόγραμμα σε σχέση με τις υπόλοιπες υλοποιήσεις που χρησιμοποιούν μόνο local locking με λιγότερα κλειδώματα.
- Το σχήμα optimistic synchronization έχει μια ενδιάμεση επίδοση, η οποία οφείλεται κυρίως στην ανάγκη της validate να διατρέξει ολόκληρη τη λίστα κάθε φορά.