25/01/2020

Συστήματα Παράλληλης Επεξεργασίας

Σειρά 3η - Τελική Αναφορά

Ομάδα: parlab30
Τσαγκαράκης Στυλιανός - 03115180
Τσάκας Νικόλαος - 03115433

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. Εξετάσαμε πως κάποια χαρακτηριστικά της αρχιτεκτονικής του συστήματος επηρεάζουν την επίδοση εφαρμογών που εκτελούνται σε αυτά και αξιολογήσαμε διάφορους τρόπους υλοποίησης κλειδωμάτων για αμοιβαίο αποκλεισμό καθώς και διάφορες τεχνικές συγχρονισμού για δομές δεδομένων.

2. Λογαριασμοί Τράπεζας

Δόθηκε ένα πολυνηματικό πρόγραμμα όπου κάθε νήμα εκτελεί ένα σύνολο πράξεων (αύξηση κατά ένα) πάνω σε ένα συγκεκριμένο στοιχείο του πίνακα που αντιπροσωπεύει τους λογαριασμούς των πελατών μια τράπεζας. Χρησιμοποιείται η βιβλιοθήκη Posix Threads (pthreads) για τη δημιουργία και διαχείριση πολλαπλών νημάτων. Χρησιμοποιείται επίσης η μεταβλητή περιβάλλοντος MT_CONF μέσω της οποίας ρυθμίζεται και ο αριθμός των νημάτων αλλά και οι πυρήνες στους οποίους θα εκτελεστούν τα πολλαπλά νήματα.

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

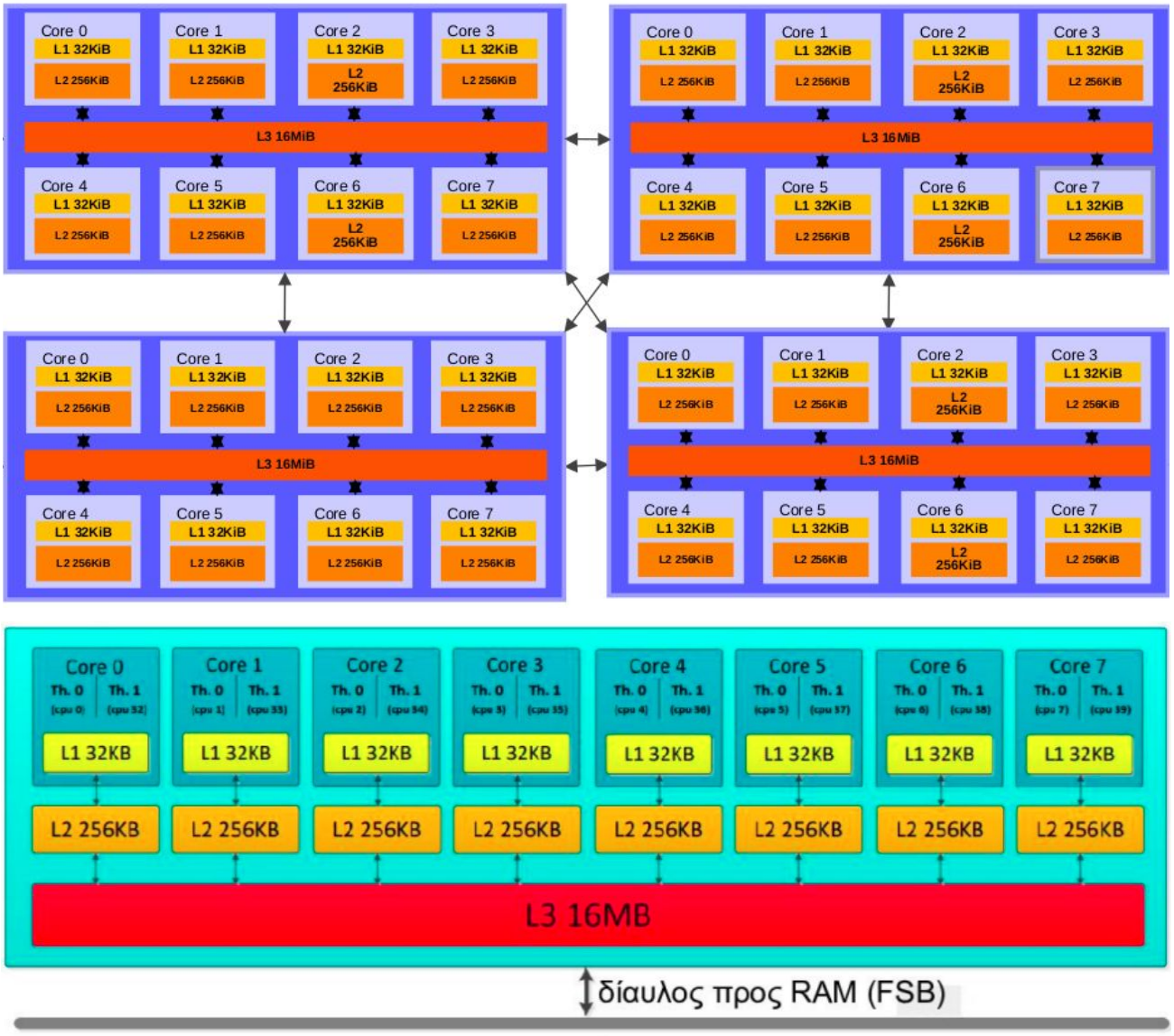
Δεν υπάρχει ανάγκη για συγχρονισμό καθώς κάθε τμήμα εκτελεί ένα σύνολο πράξεων σε διαφορετική θέση μνήμης, οπότε δεν επηρεάζονται μεταξύ τους.

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

Σίγουρα θα επηρεάζεται η ταχύτητα διότι πρέπει να υπάρχει συνέπεια στη μνήμη. Καθώς τα νήματα τροποποιούν συνεχόμενες θέσεις μνήμης είναι λογικό να υπάρχουν συνεχώς cache-to-cache transfers αφού κάθε cache block περιέχει περισσότερους από έναν “λογαριασμούς”. Αυτό αναγκάζει το ΛΣ λόγω του πρωτοκόλλου συνάφειας κρυφής μνήμης να βάζει barriers στα νήματα, περιμένοντας τη μνήμη να ενημερωθεί. Συνεπώς όσο περισσότερα νήματα, τόσο μεγαλύτερη χρήση του memory bus, δηλαδή τόσο περισσότερες cache-to-cache transfers και τόσο περισσότερα memory barriers. Άρα η επίδοση αναμένεται να μειώνεται. Παρακάτω βλέπουμε τα σχετικά διαγράμματα και σχολιασμούς πάνω σε αυτά.

Η επόμενη εικόνα μας δείχνει την οργάνωση στους επεξεργαστές του sandman και έτσι μπορούμε να αντιληφθούμε και να εξηγήσουμε καλύτερα τις καθυστερήσεις.
Αρίθμηση πυρήνων στον sandman:

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



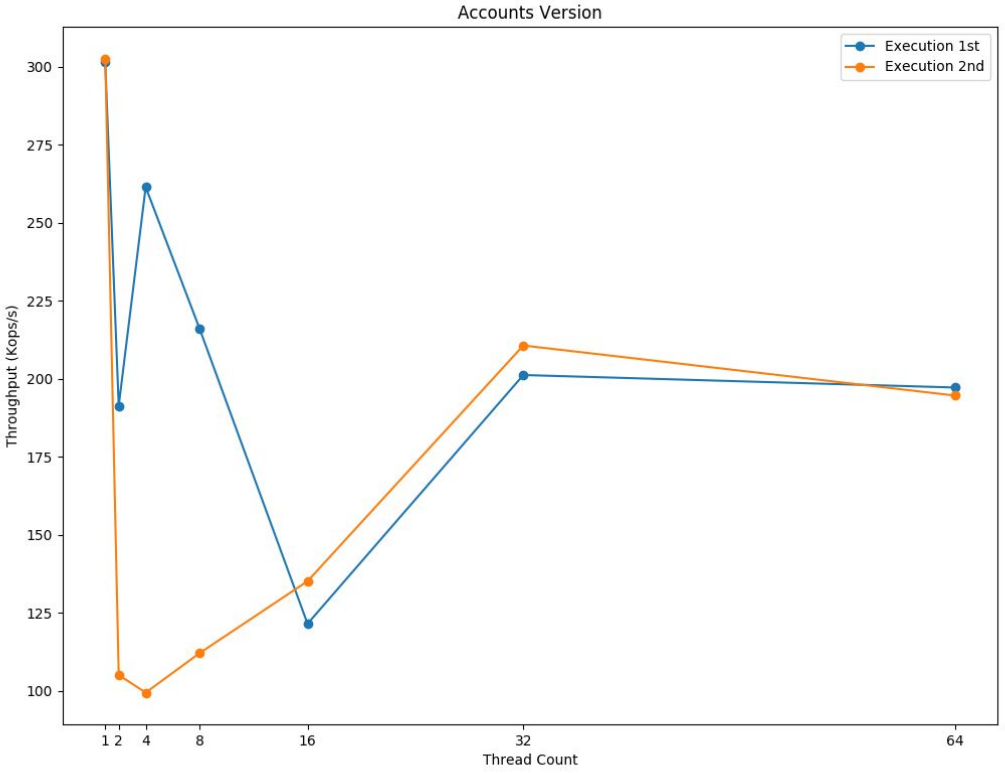
Αυτό που μπορούμε να παρατηρήσουμε είναι ότι:

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

2.3. Διαγράμματα, συμπεριφορά εφαρμογής για κάθε εκτέλεση και εξήγηση αυτής.

Εδώ βλέπουμε αρκετές αυξομειώσεις στο throughput της εφαρμογής της αρχικής έκδοσης. Βλέπουμε ότι με ένα (1) νήμα έχουμε το μεγαλύτερο throughput, κάτι που είναι λογικό καθώς δεν χρειάζονται cache-to-cache transfers.
Σημειώνεται μεγάλη πτώση στην επίδοση όταν προστίθεται ένα ακόμα νήμα καθώς χρειάζεται πολύ περισσότερες cache-to-cache transfers. Αυτό

συμβαίνει γιατί τα δυο νήματα βρίσκονται σε διαφορετικούς επεξεργαστές και επεξεργάζονται συνεχόμενες θέσεις μνήμης και συνεπώς όπως είπαμε πρέπει να ικανοποιείται το πρωτόκολλο συνέπειας κρυφής μνήμης.
Και στις δυο εκτελέσεις βλέπουμε μια άνοδο στην επίδοση όταν προστίθενται ακόμα 2 νήματα. Βλέπουμε όμως πως στην πρώτη εκτέλεση είναι μεγαλύτερη. Αυτό συμβαίνει γιατί τα νήματα βρίσκονται σε “κοντινούς” επεξεργαστές, αν όχι και στους ίδιους (αν το σύστημα υποστηρίζει Hyperthreading) και έτσι οι cache-to-cache transfers είναι πιο γρήγορες ή και ανύπαρκτες. Η λογική αυτή ακολουθείται σε όλο το υπόλοιπο διάγραμμα. Παρατηρούμε πως όταν έχουμε συνεχόμενους επεξεργαστές η επίδοση αυξάνεται για τους λόγους που αναφέραμε παραπάνω.



Υπάρχουν δυο τροποποιήσεις που θα μπορούσε κανείς να κάνει στον κώδικα. Η πρώτη είναι η εξής:

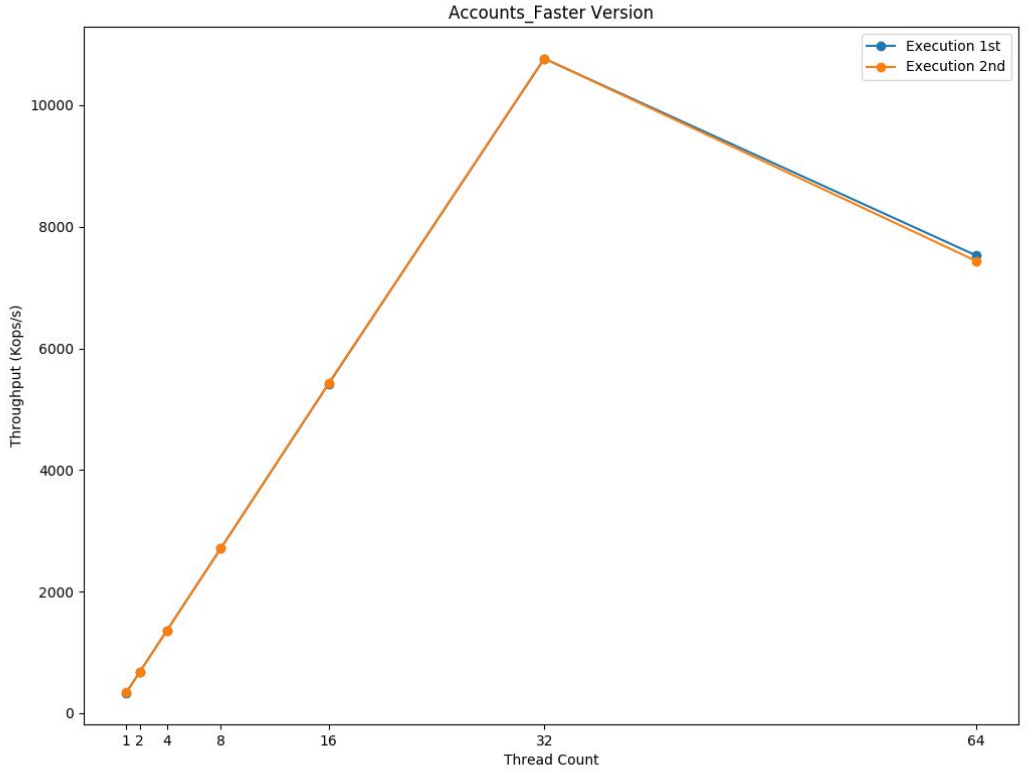
```
struct {
  unsigned int value;
  char padding_64 [64 - sizeof(unsigned int)];
} accounts[MAX_THREADS];
```

Η παρακάτω έκδοση της εφαρμογής accounts είναι μια γρηγορότερη διότι εκμεταλλευόμαστε την κρυφή μνήμη του συστήματος αφου τα νήματα δεν τροποποιούν συνεχόμενες θέσεις μνήμης.
Η τροποποίηση που κάναμε στον κώδικα είναι η εξής:

```
typedef struct {
  ...
       int offset;
  ...
} tdata_t;
...
int off = MAX_THREADS/nthreads;
threads_data[i].offset = i*(off-1);
index = mydata->offset;
...
accounts[index].value++;
...
```

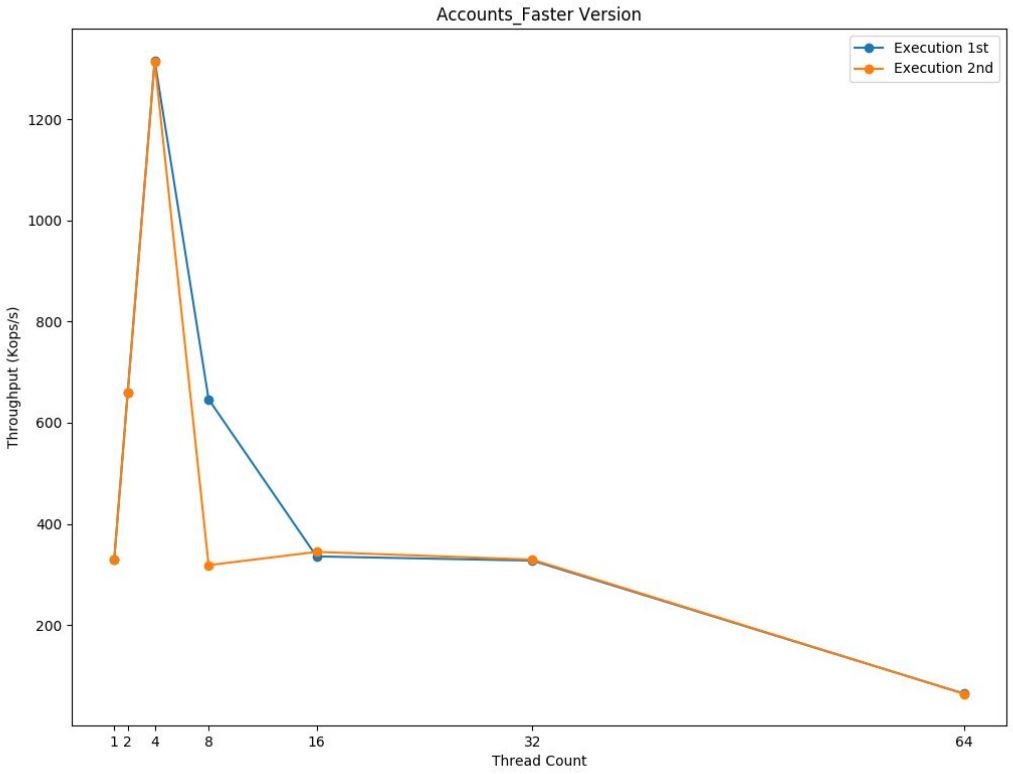
Προσθέσαμε δηλαδή ένα πεδίο offset στο struct το οποίο υπολογίζεται όπως φαίνεται παραπάνω. Έτσι κάθε νήμα γράφει σε μη συνεχόμενες θέσεις μνήμης σε σχέση με το προηγούμενο και βελτιστοποιείται η χρήση της cache.
Η τελευταία τροποποίηση φυσικά δεν ανταποκρίνεται στην πραγματικότητα διότι δεν παραμένει η αντιστοιχία στη μνήμη με τους λογαριασμούς της τράπεζας. Δοκιμάσαμε και τις δυο τροποποιήσεις.

1η Τροποποίηση:



Εδώ παρατηρούμε τρομερή βελτίωση σε σχέση με τις άλλες εκτελέσεις και πως στα 64 threads έχουμε μια μικρή πτώση στην απόδοση, λόγω της χωρητικότητας της cache.

2η Τροποποίηση:



Παρατηρούμε ότι σε μικρό αριθμό νημάτων η επίδοση ανεβαίνει κατακόρυφα, καθώς κάθε thread παίρνει διαφορετικό block και δεν υπάρχουν cache-to-cache transfers.

Παρακάτω βλέπουμε και τα στοιχεία της cache:

Cache Level	**Type**	**Size**
1	Instruction	32 KB
1	Data	32 KB
2	Unified	256 KB
3	Unified	16384 KB

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 αξιολογήσετε 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

   Τα είδη κλειδωμάτων που μελετώνται:

- **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 "../common/alloc.h"
#include <pthread.h>

struct lock_struct
{
   pthread_spinlock_t spinlock;
};

lock_t *lock_init(int nthreads)
{
   lock_t *lock;
   XMALLOC(lock, 1);
   pthread_spin_init(&lock->spinlock,
PTHREAD_PROCESS_PRIVATE);
   return lock;
}

void lock_free(lock_t *lock)
```

```
    {
        pthread_spin_destroy(&lock->spinlock);
        XFREE(lock);
    }

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

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

- **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 "../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);
```

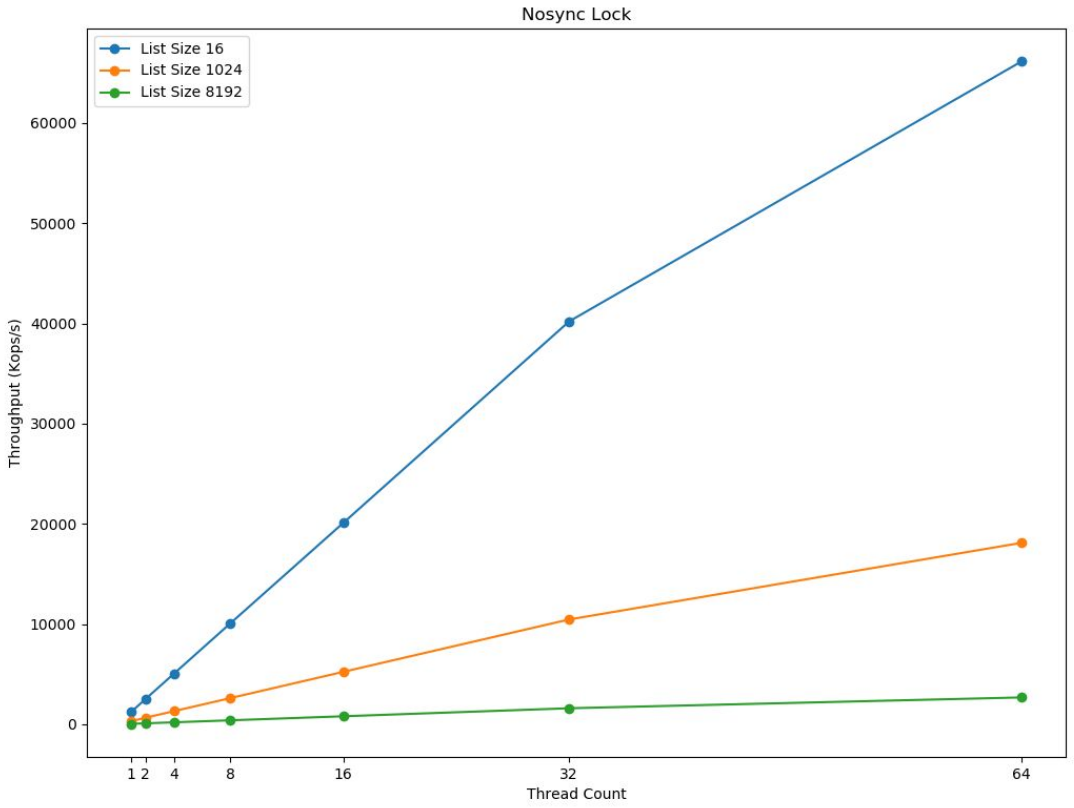
```
    lock->state = UNLOCKED;

    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) /* do nothing */ ;

        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);
}
```

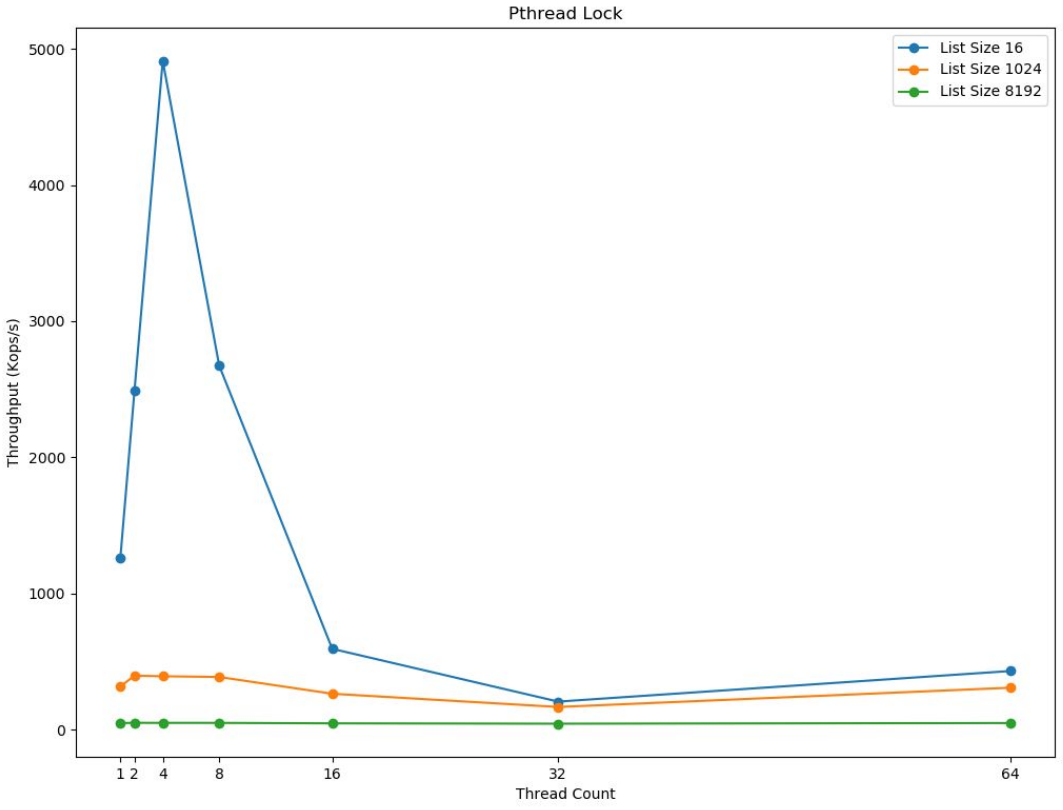
- **array_lock:** Το array-based κλείδωμα όπως περιγράφεται και στις διαφάνειες του μαθήματος.

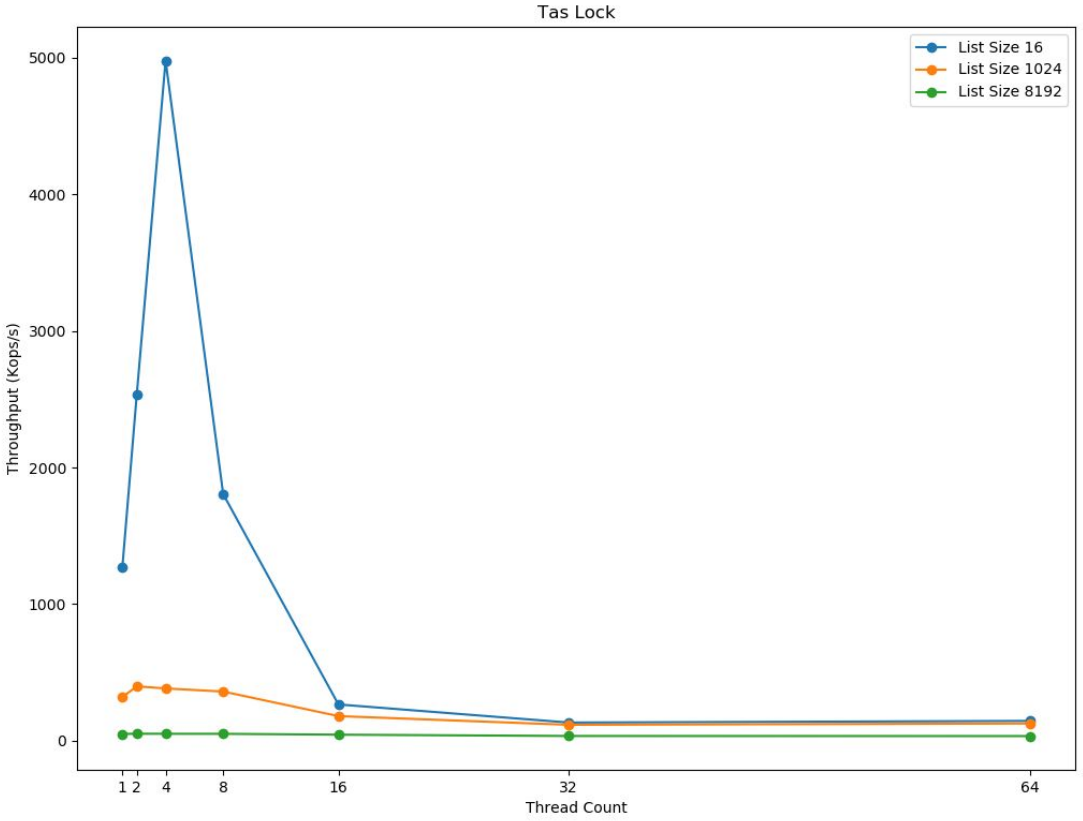
```
#include "lock.h"
#include "../common/alloc.h"

__thread int mySlotIndex;

struct lock_struct {
    int* flag;
```

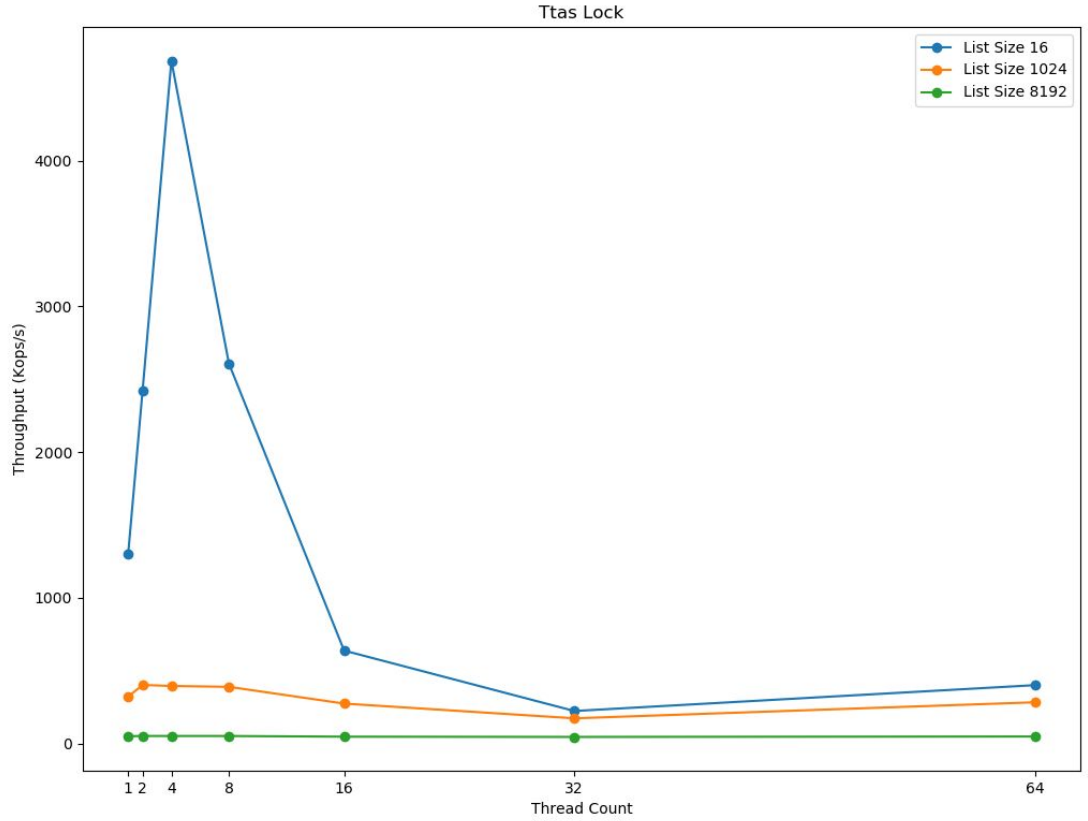
```
    int tail; // Perform only atomic ops on this
    int size;
};

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

    XMALLOC(lock, 1);
    lock->size = nthreads;
    XMALLOC(lock->flag, nthreads);
    lock->flag[0] = 1;
    for (int i=1; i<nthreads; ++i)
        lock->flag[i]=0;
    lock->tail = 0;
    return lock;
}
void lock_free(lock_t *lock)
{
    XFREE(lock->flag);
    XFREE(lock);
}

void lock_acquire(lock_t *lock)
{
    lock_t *l = lock;
    int slot = __sync_fetch_and_add(&l->tail, 1) %
l->size;
    mySlotIndex = slot;
    while(!l->flag[slot]) /* do nothing */ ;
}

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

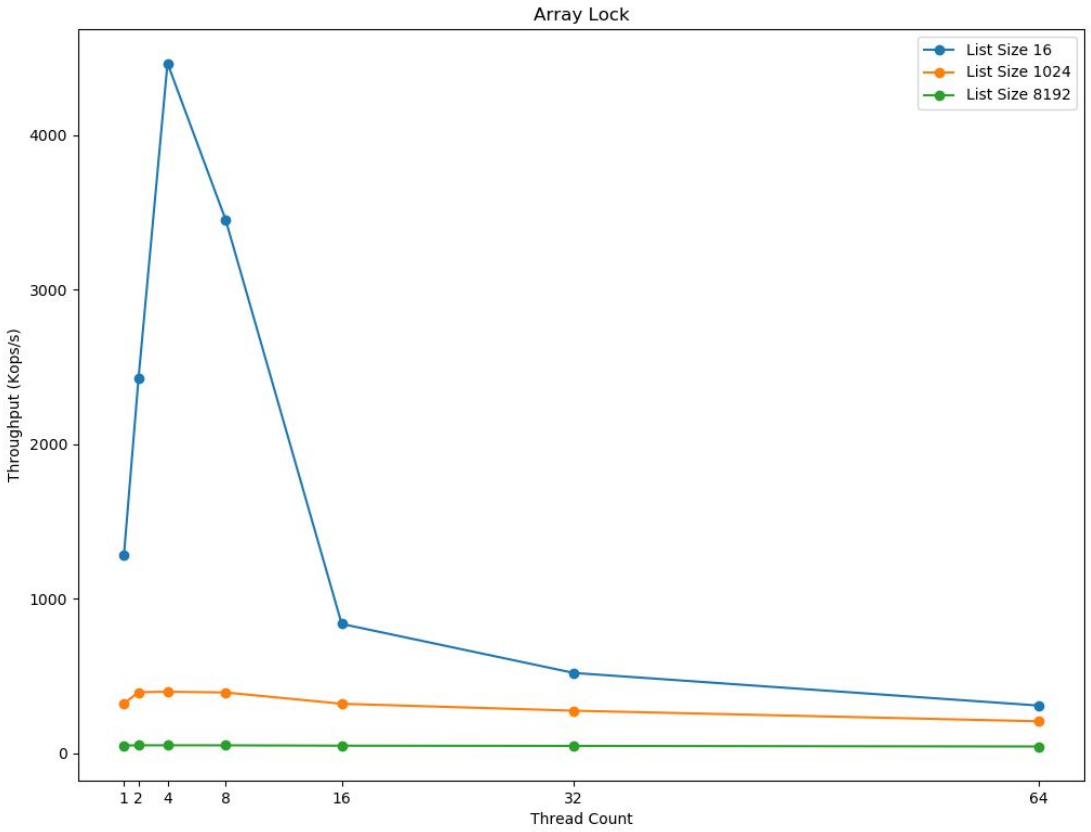
- **clh_lock**: Ένα είδος κλειδώματος που στηρίζεται στη χρήση μίας συνδεδεμένης λίστας. Αναλυτικές πληροφορίες μπορούμε να βρούμε στο Κεφάλαιο 7 του βιβλίου ”The Art of Multiprocessor Programming”. Η υλοποίησή του μας δίνεται.






Tas Lock
List Size 16
List Size 1024
List Size 8192
Throughput (Kops/s)
5000
4000
3000
2000
1000
0
1 2 4 8 16 32 64
Thread Count


Ttas Lock
List Size 16
List Size 1024
List Size 8192
Throughput (Kops/s)
4000
3000
2000
1000
0
1 2 4 8 16 32 64
Thread Count



Σχολιασμός:

1. Όσο μεγαλύτερη λίστα έχουμε τόσο περισσότερο μεγαλώνει το κρίσιμο τμήμα και συνεπώς η εκτέλεση διαρκεί περισσότερο.
2. Σε όλες τις εκτελέσεις παρατηρούμε μεγιστοποίηση του throughput στα 4 threads και List_size = 16
3. Η απόκλιση των διαφορετικών κλειδωμάτων από την υλοποίηση χωρίς συγχρονισμό είναι αρκετά μικρή και αυξάνεται ραγδαία όσο μεγαλώνει ο αριθμός των νημάτων. Αυτό συμβαίνει γιατί περισσότερα νήματα αιτούνται να μπουν στο κρίσιμο τμήμα και υπάρχει μεγαλύτερη αναμονή.
4. Ελάχιστη άνοδο σημειώνει το pthread_lock και το ttas_lock στα 64 threads σε σχέση με τα 32.
5. Επίσης το ttas_lock εμφανίζεται πιο αποδοτικό από το tas_lock όπως αναμέναμε.

Συμπεράσματα:

1. Όταν επιθυμούμε λίγα νήματα (8-) είναι κατάλληλα τα κλειδώματα: pthread, tas, ttas
2. Όταν θέλουμε περισσότερα νήματα (16+) καταλληλότερα είναι τα κλειδώματα: array, clh.
3. Φυσικά μπορεί να αυξηθεί η απόδοση κάθε κλειδώματος αν κάθε thread τροποποιεί θέσεις μνήμης ανεξάρτητα από τα υπόλοιπα.

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.

4.1 Ερωτήσεις - Ζητούμενα

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

- **Serial, No locking:** Δίνεται έτοιμη η υλοποίηση του και χρησιμοποιείται ως βάση αναφοράς για τους υπόλοιπους αλγόριθμους.

- **Fine Grain Locking:** Παρακάτω δίνεται ο κώδικας που υλοποιήσαμε και στη συνέχεια παρουσιάζονται τα διαγράμματα.

```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;
   /* other fields here? */
};

static ll_node_t *ll_node_new(int key)
{
   ll_node_t *ret;

   XMALLOC(ret, 1);
   ret->key = key;
```

```
    ret->next = NULL;
    XMALLOC(ret->lock, 1);  //Clang-Tidy gives warning about this
    pthread_spin_init(ret->lock, PTHREAD_PROCESS_PRIVATE);
    return ret;
}
static void ll_node_free(ll_node_t *ll_node)
{
    pthread_spin_destroy(ll_node->lock);
    XFREE(ll_node->lock);
    XFREE(ll_node);
}
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;
}
void ll_free(ll_t *ll)
{
    ll_node_t *next, *curr = ll->head;
    while (curr) {
        next = curr->next;
        ll_node_free(curr);
        curr = next;
    }
    XFREE(ll);
}
int ll_contains(ll_t *ll, int key)
{
    ll_node_t *curr;
    pthread_spin_lock(ll->head->lock);
    curr = ll->head;
    ll_node_t *succ;
    pthread_spin_lock(curr->next->lock);
```

```
    succ = curr->next;
    while(curr->key < key && succ->key != INT_MAX)
    {
        pthread_spin_unlock(curr->lock);
        curr = succ;
        succ = succ->next;
        pthread_spin_lock(succ->lock);
    }
    int temp_key = curr->key;
    pthread_spin_unlock(curr->lock);
    pthread_spin_unlock(succ->lock);
    return (temp_key == key);
}
int ll_add(ll_t *ll, int key)
{
    ll_node_t *pred, *succ;
    pthread_spin_lock(ll->head->lock);
    pred = ll->head;
    succ = pred->next;
    pthread_spin_lock(succ->lock);
    while(succ->key < key)
    {
        pthread_spin_unlock(pred->lock);
        pred = succ;
        succ = succ->next;
        pthread_spin_lock(succ->lock);
    }
    if (succ->key != key) {
        ll_node_t *new_node = ll_node_new(key);
        pred->next = new_node;
        new_node->next = succ;
        pthread_spin_unlock(succ->lock);
        pthread_spin_unlock(pred->lock);
        return 1;
    }
    else {
        pthread_spin_unlock(succ->lock);
        pthread_spin_unlock(pred->lock);
        return 0;
```

```
    }
}
int ll_remove(ll_t *ll, int key)
{
    ll_node_t *pred, *curr;
    pthread_spin_lock(ll->head->lock);
    pred = ll->head;
    curr = pred->next;
    pthread_spin_lock(curr->lock);
    while(curr->key < key)
    {
        pthread_spin_unlock(pred->lock);
        pred = curr;
        pthread_spin_lock(curr->next->lock);
        curr = curr->next;
    }
    if(curr->key == key)
    {
        pred->next = curr->next;
        ll_node_free(curr);
        pthread_spin_unlock(pred->lock);
        return 1;
    } else
    {
        pthread_spin_unlock(curr->lock);
        pthread_spin_unlock(pred->lock);
        return 0;
    }
}
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:** Παρακάτω δίνεται ο κώδικας που υλοποιήσαμε και στη συνέχεια παρουσιάζονται τα διαγράμματα.

```
#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;
    /* other fields here? */
};
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;
}
static void ll_node_free(ll_node_t *ll_node)
{
    pthread_spin_destroy(&(ll_node->lock));
    XFREE(ll_node);
}
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;
}
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 *curr,ll_node_t *next,ll_t *list) {
    ll_node_t *node=list->head;
    int ret=0;
    while (node->key <= curr->key) {
        if (node == curr) {
            ret= (curr->next == next);
            break;
        }
        node = node->next;
    }
    return ret;
}
int ll_contains(ll_t *ll, int key)
{
    int ret=0;
    ll_node_t *curr,*next;
    while(1){
        curr=ll->head;
        next=ll->head->next;
        while (next->key <= key) {
```

```
            curr = next;
            next = next->next;
        }
        pthread_spin_lock(&(curr->lock));
        pthread_spin_lock(&(next->lock));
        if (validate(curr,next,ll)) {
            if(next->key == key) {
                ret = 1;
            }
            pthread_spin_unlock(&(curr->lock));
            pthread_spin_unlock(&(next->lock));
            break;
        }
        pthread_spin_unlock(&(curr->lock));
        pthread_spin_unlock(&(next->lock));
    }
    return ret;
}
int ll_add(ll_t *ll, int key)
{
    int ret=0;
    ll_node_t *curr,*next;
    ll_node_t *new_node;
    while (1) {
        curr=ll->head;
        next=ll->head->next;
        while (next->key <= key) {
            curr=next;
            next=next->next;
        }
        pthread_spin_lock(&(curr->lock));
        pthread_spin_lock(&(next->lock));
        if (validate(curr,next,ll)) {
            if (next->key != key) {
                ret = 1;
                new_node=ll_node_new(key);
                new_node->next=next;
                curr->next=new_node;
            }
```

```
            else {
                ret = 0;
            }
            pthread_spin_unlock(&(curr->lock));
            pthread_spin_unlock(&(next->lock));
            break;
        }
        pthread_spin_unlock(&(curr->lock));
        pthread_spin_unlock(&(next->lock));
    }
    return ret;
}
int ll_remove(ll_t *ll, int key)
{
    int ret=0;
    ll_node_t *curr,*next;

    while(1) {
        curr=ll->head;
        next=ll->head->next;
        while (next->key <= key) {
            if (key == next->key) break;
            curr=next;
            next=next->next;
        }
        pthread_spin_lock(&(curr->lock));
        pthread_spin_lock(&(next->lock));
        if (validate(curr,next,ll)) {
            if (key == next->key){
                ret = 1;
                curr->next = next->next;
                pthread_spin_unlock(&(curr->lock));
                pthread_spin_unlock(&(next->lock));
                ll_node_free(next);
            }
            else {
                ret = 0;
                pthread_spin_unlock(&(curr->lock));
                pthread_spin_unlock(&(next->lock));
```

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

- **Lazy synchronization:** Παρακάτω δίνεται ο κώδικας που υλοποιήσαμε και στη συνέχεια παρουσιάζονται τα διαγράμματα.

```
#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;
    pthread_spinlock_t state;
    int marked;
    struct ll_node *next;
} ll_node_t;

struct linked_list {
    ll_node_t *head;
    pthread_mutex_t mymutex;
};

static ll_node_t *ll_node_new(int key)
{
    ll_node_t *ret;

    XMALLOC(ret, 1);
    ret->key = key;
    ret->next = NULL;
```

```
    int pshared;
    pthread_spin_init(&ret->state,pshared);
    ret->marked=1;

    return ret;
}

static void ll_node_free(ll_node_t *ll_node)
{
    XFREE(ll_node);
}

//Validate
int validate(ll_node_t *pred, ll_node_t *curr) {
    return pred->marked && curr->marked && pred->next==curr;
}

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;
}

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

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

```
{
    ll_node_t *curr;
    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 *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->state);
        pthread_spin_lock(&curr->state);
        if (validate(pred,curr))
        {
            if (pred->key!=key)
            {
                new_node = ll_node_new(key);
                new_node->next = curr;
                pred->next = new_node;

                ret=1;
                pthread_spin_unlock(&pred->state);
                pthread_spin_unlock(&curr->state);
                return ret;
            }
            ret=0;
```

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

int ll_remove(ll_t *ll, int key)
{
    int ret=0;
    ll_node_t *pred ,*curr;
    while (1)
    {
        pred=ll->head;
        curr=pred->next;
        while (curr->key<key)
        {
            pred=curr;curr=curr->next;
        }
        pthread_spin_lock(&pred->state);
        pthread_spin_lock(&curr->state);
        if (validate(pred,curr))
        {
            if (curr->key==key)
            {
                curr->marked=0;
                pred->next=curr->next;
                ret=1;
                pthread_spin_unlock(&pred->state);
                pthread_spin_unlock(&curr->state);
                return ret;
            }
            ret=0;
            pthread_spin_unlock(&pred->state);
            pthread_spin_unlock(&curr->state);
            return ret;
        }
```

```
        pthread_spin_unlock(&pred->state);
        pthread_spin_unlock(&curr->state);
    }
}
```

- **Non-Blocking synchronization:** Παρακάτω δίνεται ο κώδικας που υλοποιήσαμε και στη συνέχεια παρουσιάζονται τα διαγράμματα.

```
#include <stdio.h>
#include <stdlib.h> /* rand() */
#include <limits.h>
#include "../common/alloc.h"
#include "ll.h"

/*
* This is defined mainly to distinct variables declared for use
* as combinations of
* point64_ter and marking bit.
*/
typedef int64_t comb_t;

typedef struct ll_node {
    int key;
    comb_t field; // Next node point64_ter + "marked" bit
} ll_node_t;

/*
* Macros to assist in creating and extracting values
* using the combined type. The 2 last ones get an entire
* node and do the extraction.
*/
#define LSB_MASK 0x0000000000000001
#define BIT_MASK 0xfffffffffffffffe
#define COMBINE(P, B) ((comb_t)((int64_t)P | B))
#define GETNEXTPOINTER(N) ((ll_node_t *)(N->field & BIT_MASK))
#define GETMARKEDBIT(N) (N->field & LSB_MASK)

// Similar to the get() method of the java atomic reference
class
```

```
static ll_node_t *getnextandcheck(ll_node_t *n, int64_t *marked)
{
   // Get the field out first in order to perform both
operations on the same value.
   comb_t f = n->field;
   marked[0] = f & LSB_MASK;

   return (ll_node_t *)(f & BIT_MASK);
}

/*
* Similar to the compareAndSet() method in Java, checks if
* node->field == expected and on
* success sets it to update. Returns 1 on sucess.
*/
static int64_t compare_and_set(ll_node_t *node, comb_t expected,
comb_t update)
{
   int val = __sync_bool_compare_and_swap(&node, expected,
update);
   return val;
}

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

// For use in the find() function
typedef struct window
{
   ll_node_t *start;
   ll_node_t *end;
} window_t;

static window_t *find(ll_t *ll, int key)
{
   ll_node_t *pred, *curr, *succ;
   int64_t marked[1] = {0};
```

```
retry: while (1)
    {
        pred = ll->head;
        curr = GETNEXTPOINTER(pred);
        while (1)
        {
            // Possibly replacing function body here is faster
            succ = getnextandcheck(curr, marked);
            while (marked[0])
            {
                int64_t snip  =
__sync_bool_compare_and_swap(&pred->field, COMBINE(curr,0),
COMBINE(succ,0));

                if(!snip) goto retry;
                curr = succ;
                succ = getnextandcheck(curr, marked);
            }

            if (curr->key >= key)
            {
                window_t *ret;
                XMALLOC(ret, 1);
                ret->start = pred;
                ret->end = curr;
                return ret;
            }
            pred = curr;
            curr = succ;
        }
    }
}
static ll_node_t *ll_node_new(int key)
{
    ll_node_t *ret;

    XMALLOC(ret, 1);
    ret->key = key;
    ret->field = 1;
```

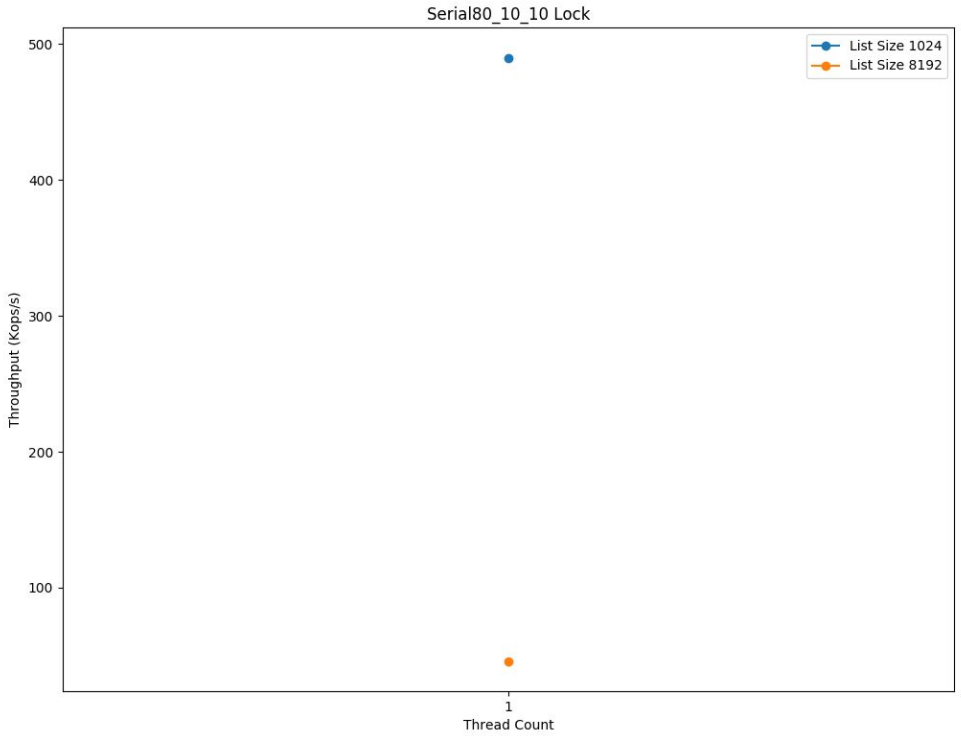
```
    return ret;
}
static void ll_node_free(ll_node_t *ll_node)
{
    XFREE(ll_node);
}
ll_t *ll_new()
{
    ll_t *ret;

    XMALLOC(ret, 1);
    ret->head = ll_node_new(-1);
    ll_node_t *last = ll_node_new(INT_MAX);
    ret->head->field = COMBINE(last, 0);
    last->field = COMBINE(NULL, 0);

    return ret;
}
void ll_free(ll_t *ll)
{
    ll_node_t *next, *curr = ll->head;
    while (curr) {
        next = GETNEXTPOINTER(curr);
        ll_node_free(curr);
        curr = next;
    }
    XFREE(ll);
}

int ll_contains(ll_t *ll, int key)
{
    int64_t marked[1] = {0};
    ll_node_t *curr = ll->head;
    while (curr->key < key)
    {
        curr = GETNEXTPOINTER(curr);
    }
```

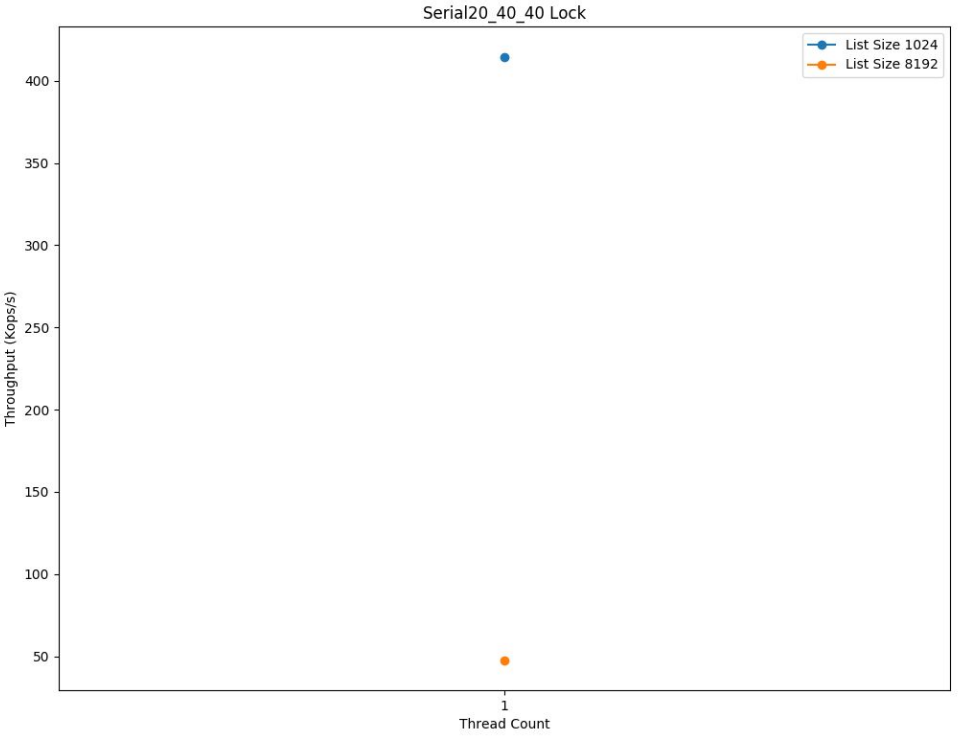
```
    // *succ is defined in lesson slides, but never used
    ll_node_t *succ = getnextandcheck(curr, marked);
    int val = (curr->key == key && !marked[0]);
    return val;

    // Alternatively:
    // return (curr->key == key && !GETMARKEDBIT(curr))
}

int ll_add(ll_t *ll, int key)
{
    // Lesson slides define a bool that is never used here:
    // int64_t splice
    while (1)
    {
        window_t *window = find(ll, key);
        ll_node_t *pred = window->start;
        ll_node_t *curr = window->end;
        if (curr->key == key)
        {
            return 0;
        }
        else
        {
            ll_node_t *node = ll_node_new(key);
            node->field = COMBINE(curr, 0);

            int val = __sync_bool_compare_and_swap(&pred->field,
COMBINE(curr,0), COMBINE(node,0));
            return val;
        }
    }
}

int ll_remove(ll_t *ll, int key)
{
    int64_t snip;
    while (1)
    {
```

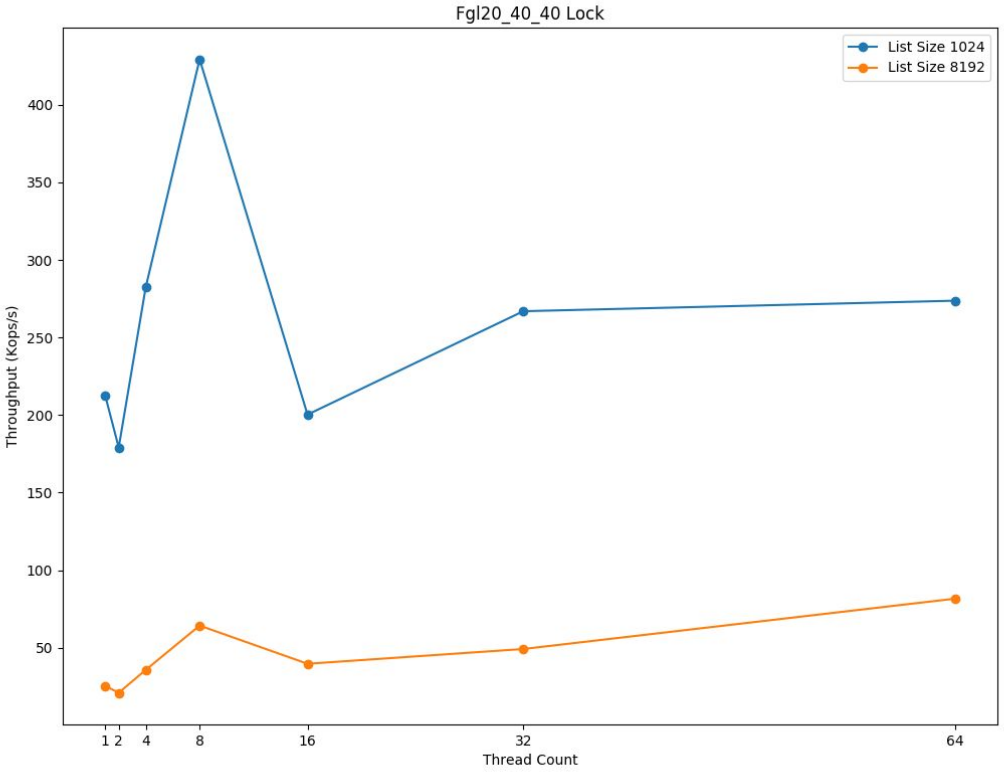
```
        window_t *window = find(ll, key);
        ll_node_t *pred = window->start;
        ll_node_t *curr = window->end;
        if (curr->key != key)
        {
            return 0;
        }
        else
        {
            ll_node_t *succ = GETNEXTPOINTER(curr);

            snip = __sync_bool_compare_and_swap(&curr->field,
COMBINE(succ,0), COMBINE(succ,1));
            if(!snip) continue;

            __sync_bool_compare_and_swap(&pred->field,
COMBINE(curr,0), COMBINE(succ,0));

            return 1;
        }
    }
}
```

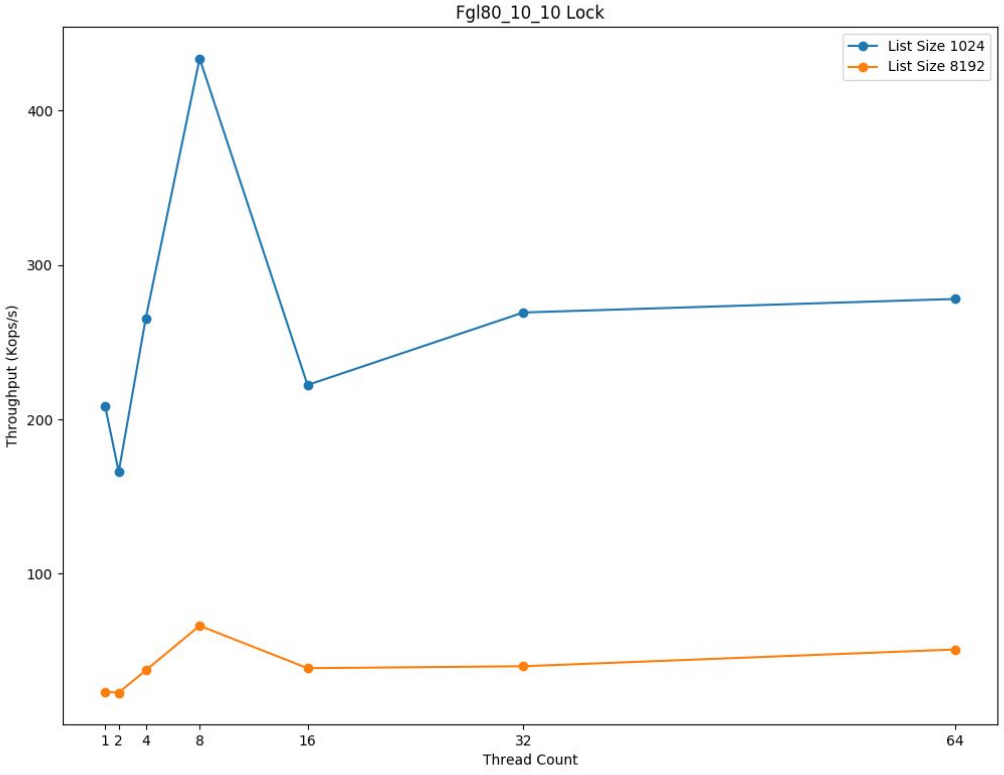
2. Εκτελέστε την εφαρμογή για όλες τις διαφορετικές υλοποιήσεις λίστας. Εκτελέστε για 1, 2, 4, 8, 16, 32, 64 νήματα, για λίστες μεγέθους 1024 και 8192 και για συνδυασμούς λειτουργιών 80-10-10 και 20-40-40. Παρουσιάστε τα αποτελέσματά σας σε διαγράμματα και εξηγήστε την συμπεριφορά της εφαρμογής για κάθε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

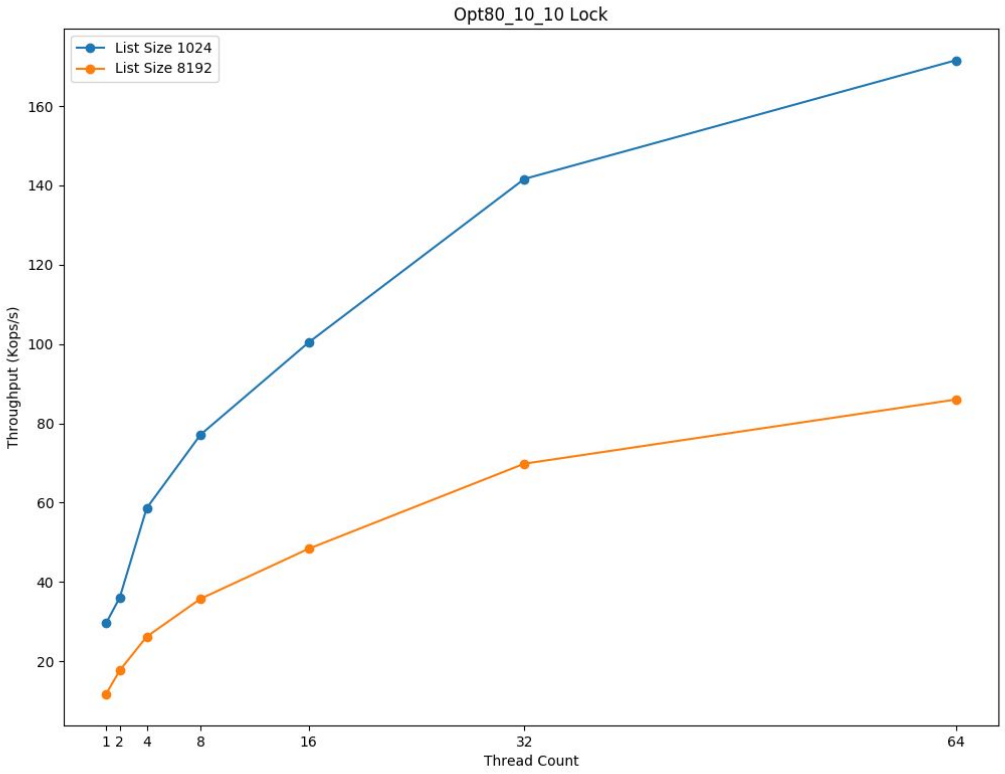
   Σημείωση: σε όλες τις εκτελέσεις θα θέσετε κατάλληλα την μεταβλητή περιβάλλοντος MT_CONF ώστε τα νήματα να καταλαμβάνουν διαδοχικούς πυρήνες, π.χ. τα 16 νήματα εκτελούνται στους πυρήνες 0-15.

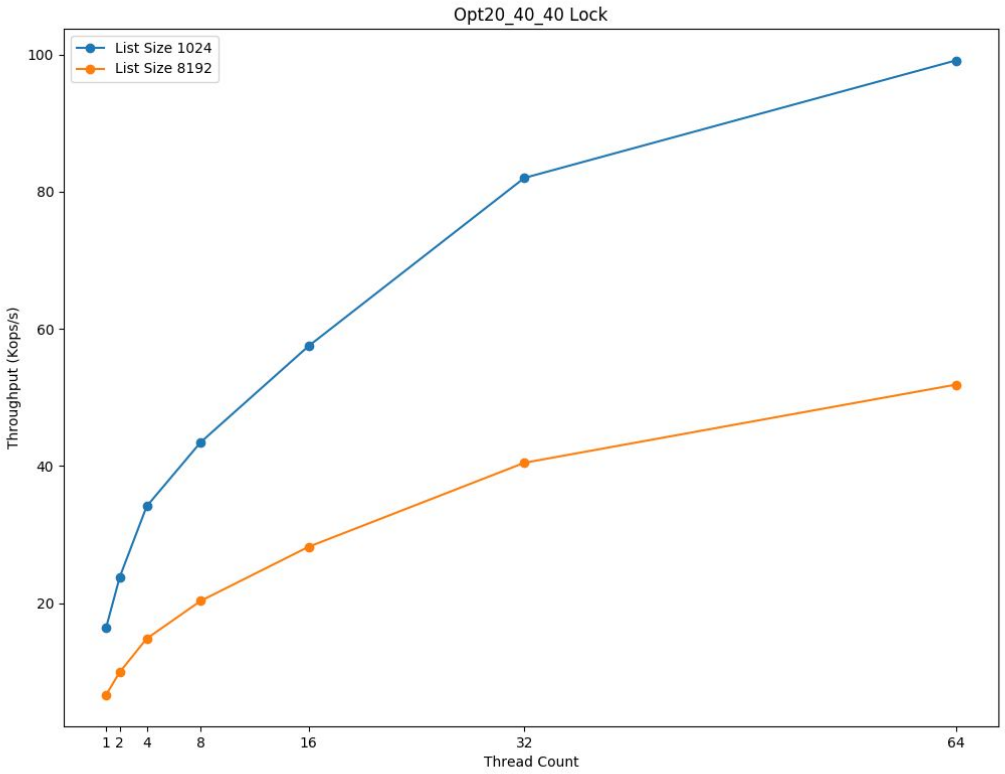

Serial80_10_10 Lock
List Size 1024
List Size 8192
500
400
300
200
100
Throughput (Kops/s)
1
Thread Count

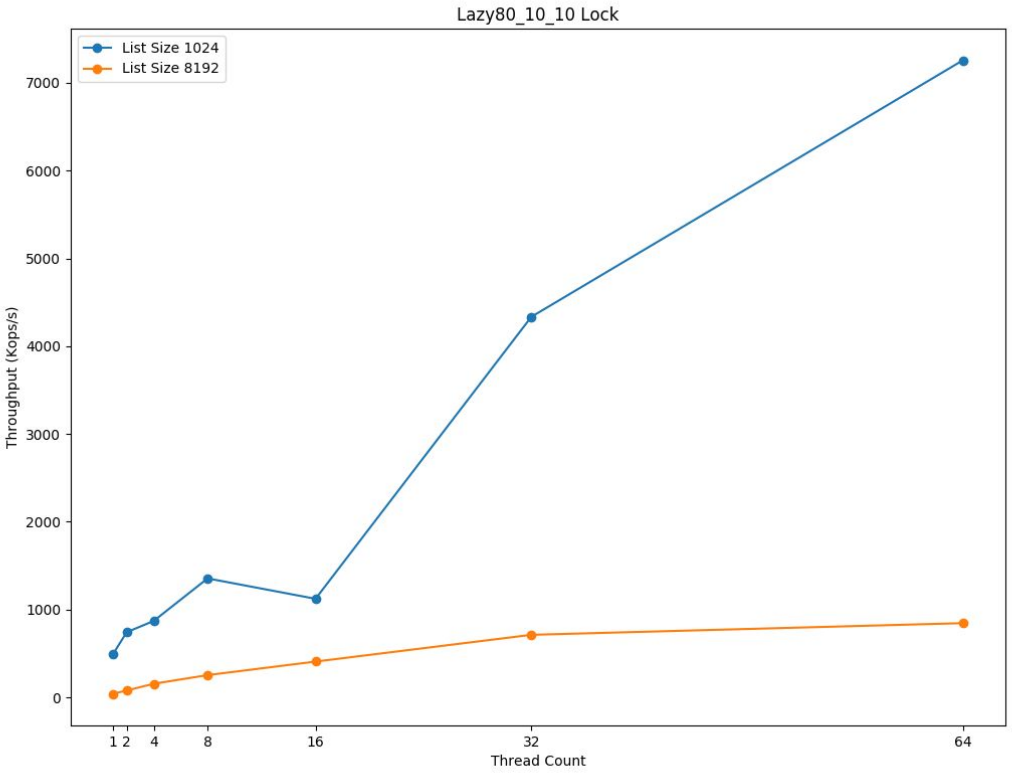

Serial20_40_40 Lock
List Size 1024
List Size 8192
400
350
300
250
200
150
100
50
Throughput (Kops/s)
1
Thread Count

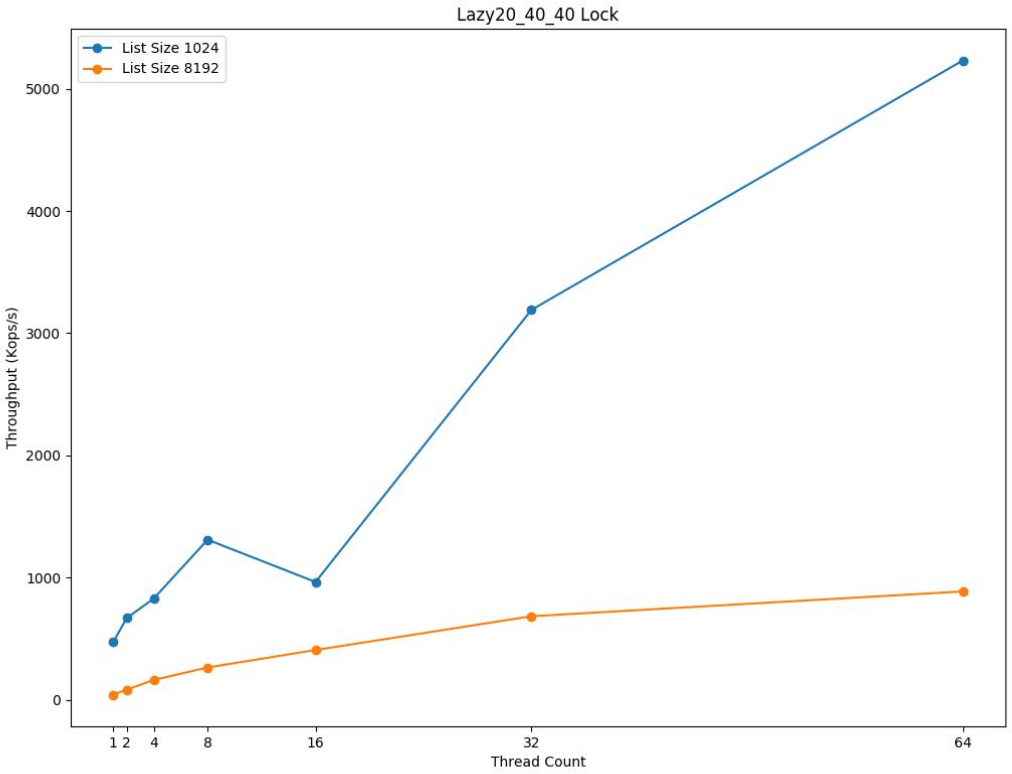

Fgl20_40_40 Lock
List Size 1024
List Size 8192
Throughput (Kops/s)
400
350
300
250
200
150
100
50
1 2 4 8 16 32 64
Thread Count

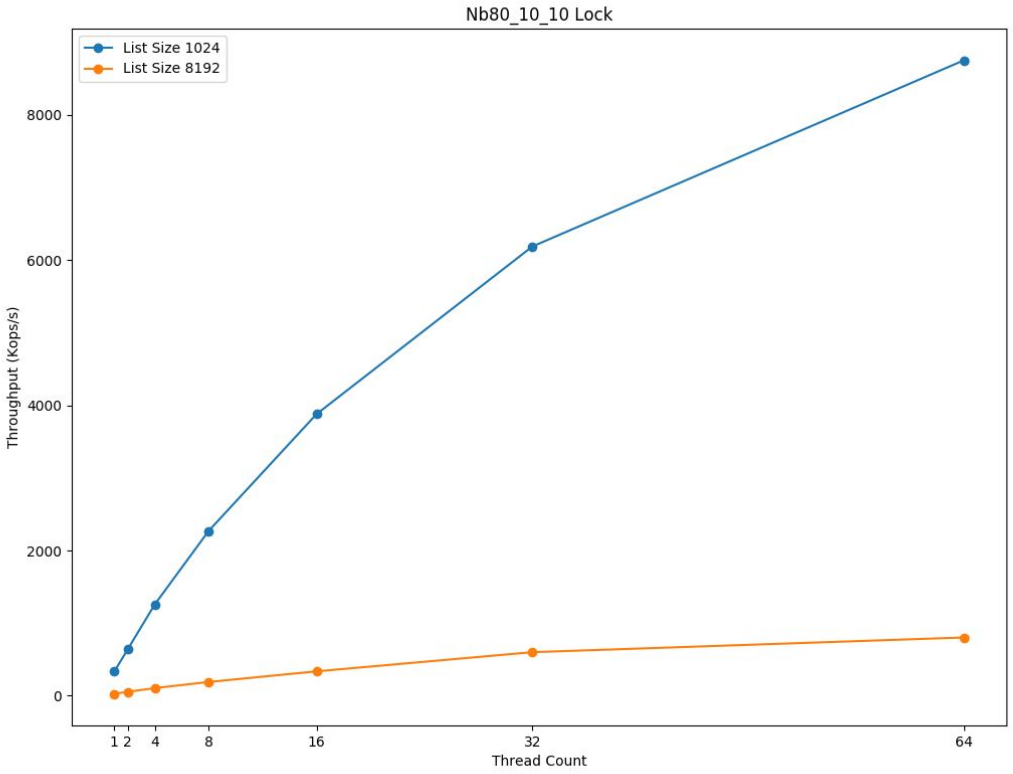

Fgl80_10_10 Lock
List Size 1024
List Size 8192
Throughput (Kops/s)
400
300
200
100
1 2 4 8 16 32 64
Thread Count

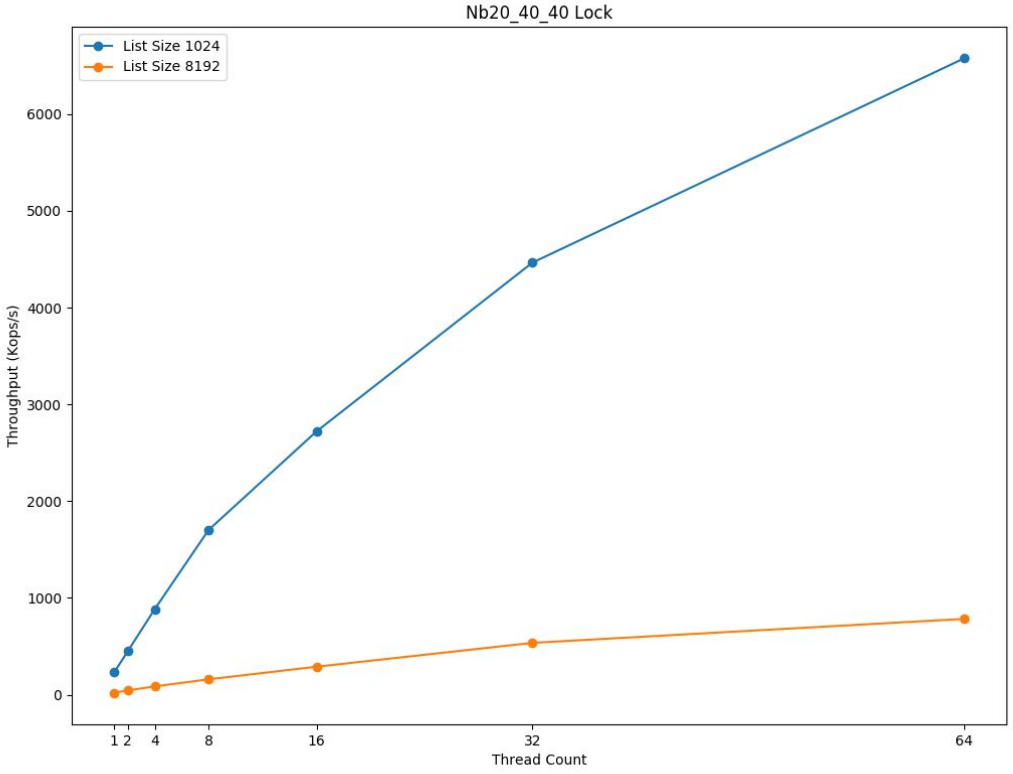

Opt80_10_10 Lock
List Size 1024
List Size 8192
Throughput (Kops/s)
20
40
60
80
100
120
140
160
1 2 4 8 16 32 64
Thread Count


Opt20_40_40 Lock
List Size 1024
List Size 8192
Throughput (Kops/s)
20
40
60
80
100
1 2 4 8 16 32 64
Thread Count


Lazy80_10_10 Lock
List Size 1024
List Size 8192
Throughput (Kops/s)
0
1000
2000
3000
4000
5000
6000
7000
1 2 4 8 16 32 64
Thread Count


Lazy20_40_40 Lock
List Size 1024
List Size 8192
Throughput (Kops/s)
0
1000
2000
3000
4000
5000
1 2 4 8 16 32 64
Thread Count


Nb80_10_10 Lock
List Size 1024
List Size 8192
Throughput (Kops/s)
0
2000
4000
6000
8000
1 2 4 8 16 32 64
Thread Count


Nb20_40_40 Lock
List Size 1024
List Size 8192
Throughput (Kops/s)
0
1000
2000
3000
4000
5000
6000
1 2 4 8 16 32 64
Thread Count

Σχολιασμός:

1. Βλέπουμε πως ισχύει η κατάταξη (από το γρηγορότερο στο πιο αργό):
    a. Non-blocking
    b. Lazy
    c. Optimistic
    d. Fine grain

    Αυτό που βλέπουμε είναι λογικό καθώς όσο πιο ψηλά στη λίστα είναι μια μέθοδος τόσο λιγότερα κλειδώματα έχει.
2. Σε όλες τις υλοποιήσεις το throughput ανεβαίνει όπως αναμέναμε, όσο αυξάνονται τα νήματα. Εκτός από την FGL καθώς όσο αυξάνονται τα νήματα υπάρχει μεγαλύτερη αναμονή στο κρίσιμο τμήμα αφού ακόμα και για να ελεγχθεί η λίστα πρέπει να κλειδώνεται με hand-over-locking, κάτι που κοστίζει αρκετά.
3. Φυσικά όσο το μέγεθος της λίστας μεγαλώνει, τόσο πέφτει το throughput όπως είναι λογικό.