Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών Ε.Μ.Π.

ΣΥΣΤΗΜΑΤΑ ΠΑΡΑΛΛΗΛΗΣ ΕΠΕΞΕΡΓΑΣΙΑΣ

2016-2017

Άσκηση 5

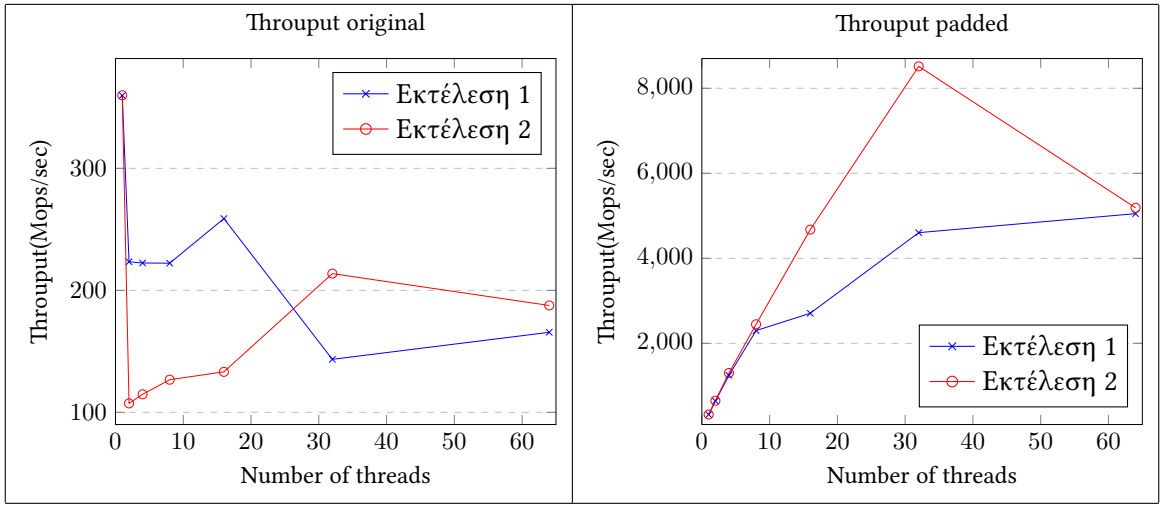
Θέματα Συγχρονισμού σε Σύγχρονα Πολυπύρηνα Συστήματα

Βαρηά Χρυσούλα - 03112105
Κουτσανίτη Ειρήνη - 03112135

Ο ζητούμενος κώδικας παρουσιάζεται στο τέλος της αναφοράς.

Λογαριασμοί Τράπεζας

Στην εικόνα 1 παρουσιάζονται τα αποτελέσματα της αρχικής (δοσμένης) και της βελτιωμένης υλοποίησης. Για να αξιοποιηθεί η ανεξαρτησία των δεδομένων ανάμεσα στα νήματα έχει προστεθεί ένα επιπλέον πεδίο στη δομή που κρατάει την τιμή value των λογαριασμών.



Εικόνα 1: Throuput αρχικής υλοποίησης και βελτιωμένης.

Απαντήσεις Ερωτήσεων - Παρατηρήσεις:

- Στη βασική υλοποίηση της εφαρμογής δεν απαιτείται συγχρονισμός μεταξύ των νημάτων, καθώς κάθε νήμα αλλάζει την τιμή value στη θέση tid της δομής accounts. Η τιμή tid είναι διαφορετική για κάθε νήμα, επομένως αυτό αλλάζει διαφορετική μεταβλητή (δηλαδή δεν υπάρχουν race conditions και εξαρτήσεις μεταξύ των ενεργειών των νημάτων).

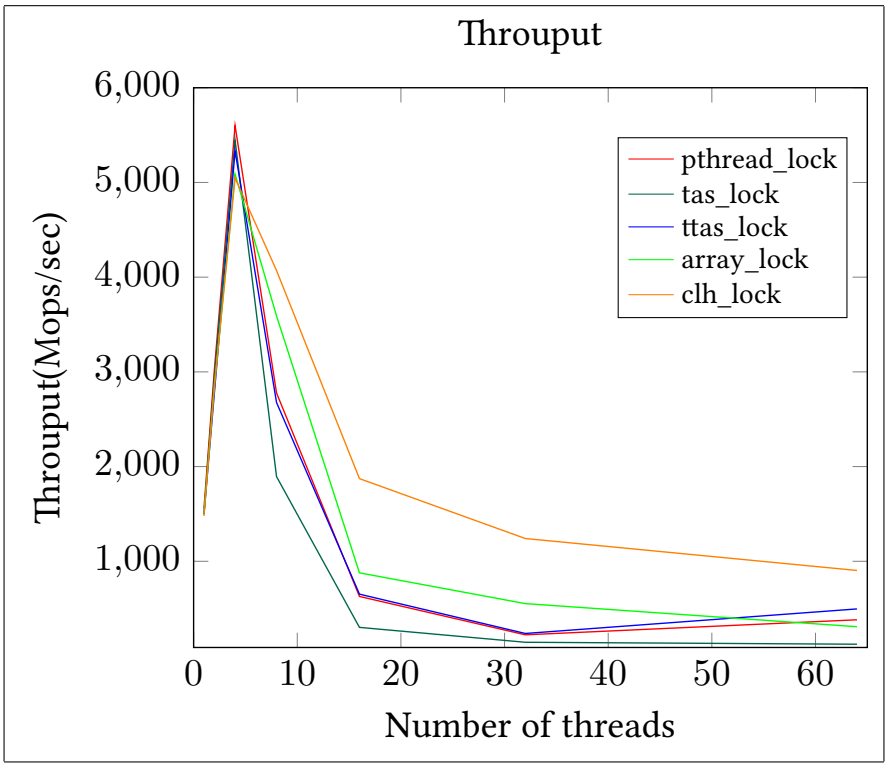
- Με την αύξηση του πλήθους των νημάτων θεωρητικά δεν πρέπει να παρουσιαστεί κάποια αλλαγή στην επίδοση, καθώς τα νήματα συγχρονίζονται μια φορά πριν αρχίσουν να αλλάζουν τις τιμές της δομής accounts και οι αλλαγές αυτές είναι ανεξάρτητες μεταξύ τους. Ωστόσο η μείωση στο throughput μπορεί να οφείλεται στην αρχιτεκτονική του sandman, καθώς για περισσότερα νήματα απαιτούνται επεξεργαστικές μονάδες από διαφορετικά numa nodes. Όταν σε ένα υποσύστημα (numa node) ένα νήμα αλλάξει κάποιο δεδομένο στη δομή accounts, η εγγραφή με το συγκεκριμένο δεδομένο γίνεται invalid στη cache των υπόλοιπων κόμβων συστημάτων. Αν ένα νήμα από άλλον υπολογιστικό κόμβο επιθυμεί να προσπελάσει γειτονικά δεδομένα (τα οποία βρίσκονται στο ίδιο block με την τιμή που άλλαξε), ανάλογα με το πρωτόκολλο cache coherence, μπορεί να προκληθεί αίτημα μεταφοράς των έγκυρων δεδομένων. Αν τα αιτήματα είναι αρκετά σε πλήθος δημιουργείται συμφόρηση στο bus, η οποία αυξάνει το συνολικό latency και μειώνει την επίδοση εκτέλεσης του προγράμματος.

- Επίσης από τα διαγράμματα φαίνεται ότι ενώ το πλήθος των νημάτων είναι ίδιο στις δύο εκτελέσεις, το αντίστοιχο throughput διαφέρει. Συγκεκριμένα η εκτέλεση 1 παρουσιάζει μεγαλύτερο throughput για πλήθος νημάτων έως 16, ενώ η εκτέλεση 2 για 32 και 64 νήματα. Αυτό μάλλον οφείλεται στον τρόπο, με τον οποίο διαμοιράζονται τα νήματα στα επιμέρους numa

nodes. Σε συνδυασμό με τις παραπάνω παρατηρήσεις, συμπεραίνουμε ότι αν νήματα διαφορετικών υποσυστημάτων μεταβάλλουν τιμές της δομής, οι οποίες βρίσκονται στο ίδιο block, η επίδοση της εκτέλεσης δεν είναι υψηλή, λόγω των cache misses και της συμφόρησης στο bus (για πλήθος νημάτων μεγαλύτερο από 16 η συμφόρηση στο bus μπορεί να είναι εντονότερη με αποτέλεσμα η εκτέλεση 2 να παρουσιάζει καλύτερη επίδοση από την εκτέλεση 1).

Η υλοποίηση των επόμενων ζητούμενων έγινε με βάση τις αντίστοιχες διαφάνειες της διάλεξης, επομένως δεν παρουσιάζεται αναλυτική εξήγηση του κώδικα.

Αμοιβαίος Αποκλεισμός - Κλειδώματα



Εικόνα 2: Throuput για μέγεθος λίστας 16.

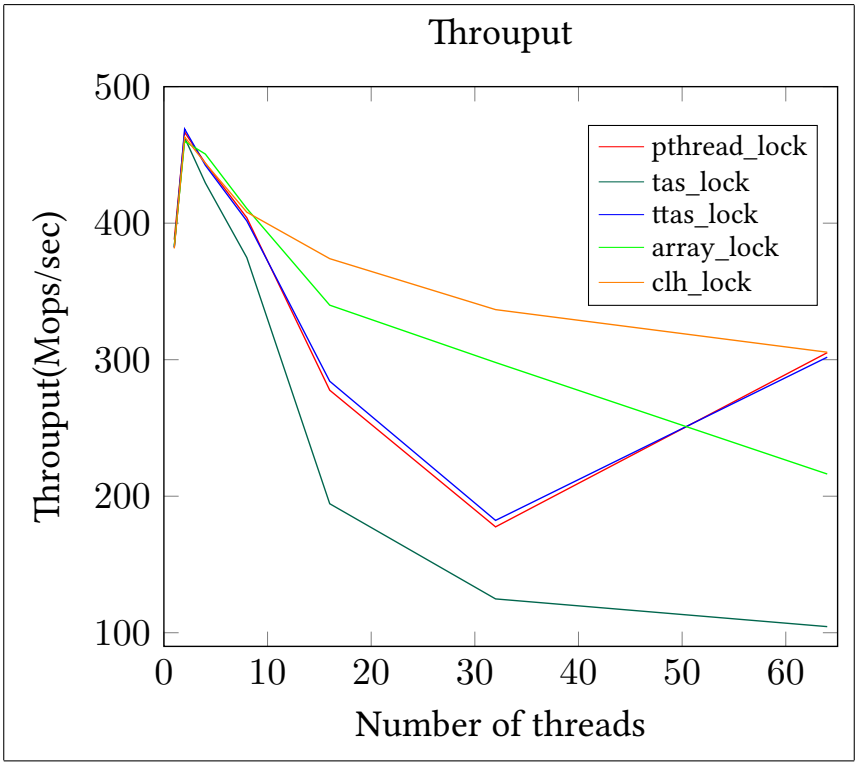
Στις εικόνες 2, 3 και 4 παρουσιάζονται τα αποτελέσματα των μετρήσεων.

Απαντήσεις Ερωτήσεων - Παρατηρήσεις:

- Αρχικά παρατηρείται ότι με την αύξηση του μεγέθους της λίστας, μειώνεται το συνολικό throughput των εντολών. Αυτό μπορεί να οφείλεται στην αύξηση του χρόνου αναζήτησης των στοιχείων μέσα στη λίστα και ίσως στον τρόπο, με τον οποίο αποθηκεύονται τα δεδομένα στις μονάδες numa nodes του sandman.

- Επίσης η αύξηση του πλήθους των νημάτων προκαλεί περισσότερα conflicts ως προς την απόκτηση του lock της δομής, το οποίο μπορεί να οδηγεί σε αυξημένη κίνηση στο bus και latency/χρόνο αναμονής των διεργασιών (άρα μικρότερο throughput).

- Όπως είναι αναμενόμενο όλες οι εκδόσεις κλειδωμάτων παρουσιάζουν μικρό throughput συγκριτικά με την υλοποίηση χωρίς κλειδώματα, το οποίο εξηγείται λόγω της αύξησης του χρό-



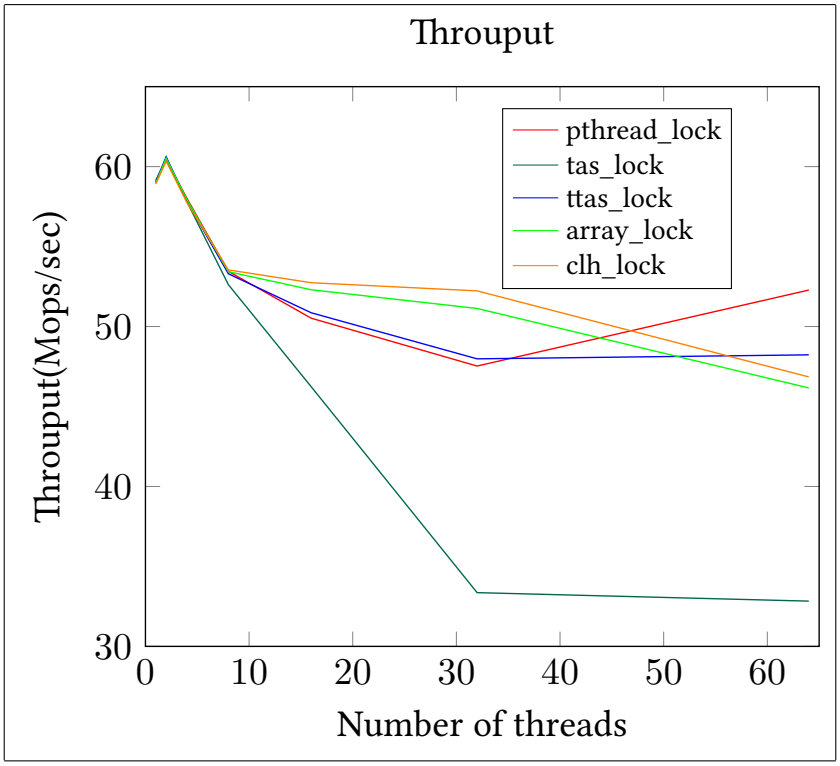
Εικόνα 3: Throuput για μέγεθος λίστας 1024.

νου αναμονής για την απόκτηση του lock. Το φαινόμενο γίνεται εντονότερο με την αύξηση του πλήθους των νημάτων.

- Στις περισσότερες περιπτώσεις η υλοποίηση clh_lock παρουσιάζει την καλύτερη επίδοση. Επίσης η έκδοση ttas_lock φαίνεται να είναι καλύτερη της tas_lock, καθώς στην πρώτη περίπτωση επιχειρείται εγγραφή στη θέση του lock μόνο όταν αυτό φαίνεται να έχει αποδεσμευτεί (σε αντίθεση με την δεύτερη υλοποίηση στην οποία γίνονται επαναλαμβανόμενες εγγραφές). Αυτό έχεις ως αποτέλεσμα να περιορίζονται τα invalidations της εγγραφής του lock στις caches των numa nodes, άρα να μειώνεται η συμφόρηση στο bus και το αντίστοιχο latency. Ακόμα η υλοποίηση του pthread_lock παρουσιάζει παρόμοια επίδοση με την έκδοση ttas_lock.
- Σε όλες τις παραπάνω υλοποιήσεις επισημαίνεται ότι οι επεξεργαστικές μονάδες παραμένουν κατειλημμένες, όση ώρα τα νήματα διεκδικούν το lock της δομής, ενώ η απόδοση τους επηρεάζεται τόσο από τον αριθμό των νημάτων όσο και από το μέγεθος του κρίσιμου τμήματος.

Τακτικές Συγχρονισμού για δομές δεδομένων

Οι αλγόριθμοι έχουν υλοποιηθεί με τον τρόπο που περιγράφεται στις διαφάνειες. Η μόνη διαφορά είναι ότι σε όλες τις υλοποιήσεις παραλείπεται το lock στη συνάρτηση contains. Αν η contains επιστρέψει θετικό αποτέλεσμα (ότι δηλαδή υπάρχει το δεδομένο στη λίστα), ενώ το στοιχείο μόλις αφαιρέθηκε από κάποιο άλλο νήμα,το αποτέλεσμα της κλήσης είναι συνεπές, καθώς δύο νήματα μπορούν να τρέχουν παράλληλα και οποιαδήποτε σειρά εκτέλεσης των remove και contains είναι αποδεκτή (δηλαδή θεωρούμε ότι η contains εκτελέστηκε και ολοκληρώθηκε πριν την αλλαγή της λίστας). Αντίστοιχες είναι οι παρατηρήσεις αν έγινε κάποια προσθήκη δεδομένου, την οποία δεν πρόλαβε η contains. Σε αυτές τις περιπτώσεις το αποτέλεσμα της συγκεκριμένης συνάρτησης μπορεί να



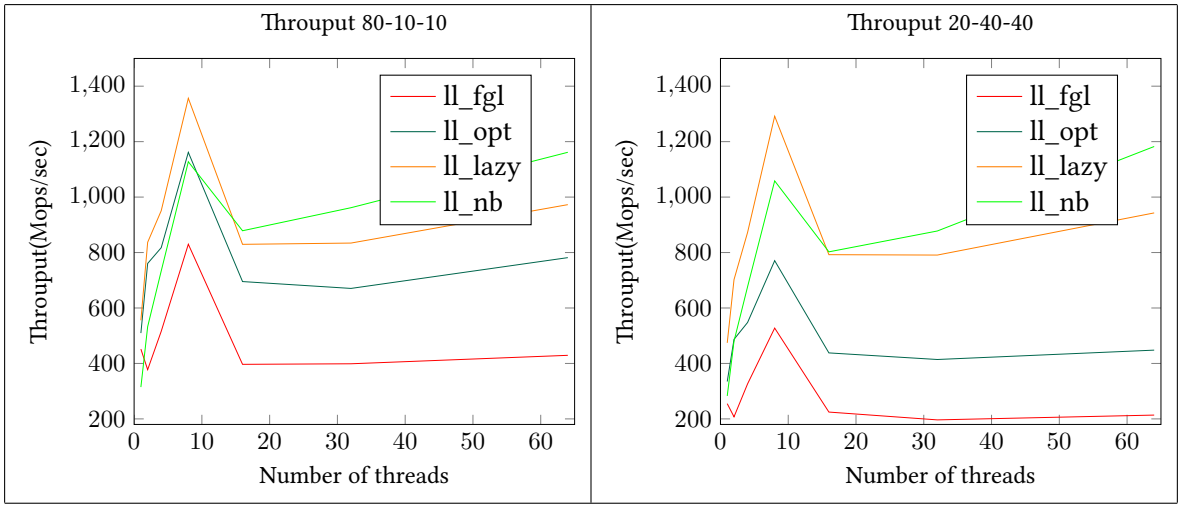
Εικόνα 4: Throuput για μέγεθος λίστας 8192.

θεωρηθεί συνεπές/σωστό (ανεξαρτήτως αποτελέσματος της επιστροφής κλήσης). Από τα παραπάνω κρίνεται ότι δεν χρειάζεται να χρησιμοποιηθεί lock.

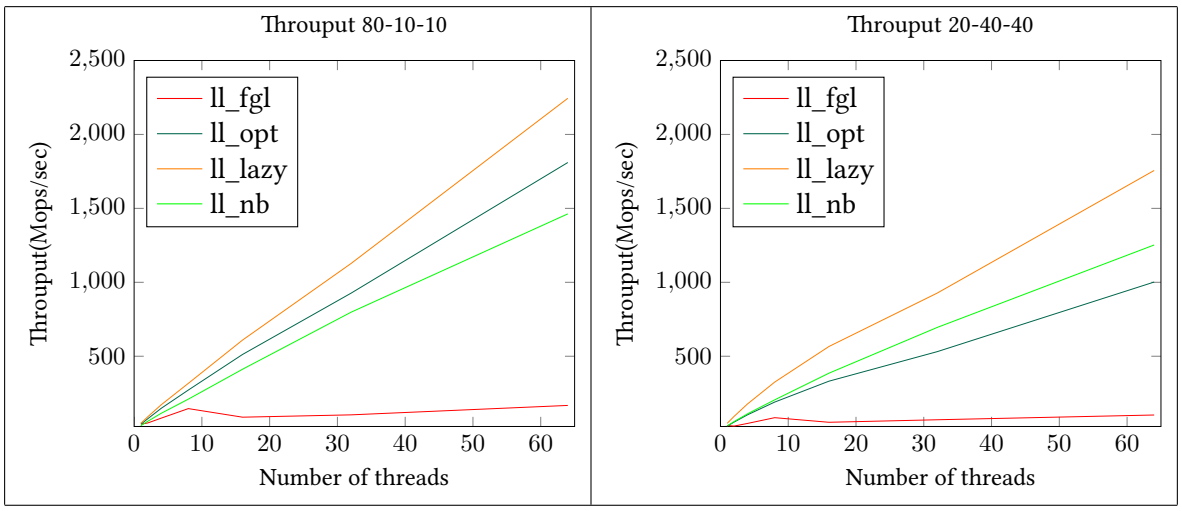
Στις εικόνες 5 και 6 παρουσιάζονται τα αποτελέσματα των μετρήσεων.

Απαντήσεις Ερωτήσεων - Παρατηρήσεις:

- Αρχικά όπως παρατηρήθηκε προηγουμένως η αύξηση του μεγέθους της λίστας προκαλεί μείωση του throughput.
- Οι υλοποιήσεις Lazy και Non-Blocking synchronization παρουσιάζουν την καλύτερη επίδοση σε όλες τις περιπτώσεις. Συγκεκριμένα για πλήθος νημάτων μικρότερο του 16 η Lazy synchronization έχει μεγαλύετρο throughput από ότι η Non-Blocking υλοποίηση, ενώ για πλήθος νημάτων μεγαλύτερο ισχύει το αντίστροφο. Η Non-Blocking έκδοση δεν χρησιμοποιεί κλείδωμα για τις υλοποιήσεις των contains, add και remove, αλλά βασίζεται σε επιτυχή εκτέλεση μέσω επαναλαμβανόμενων προσπαθειών. Αυτό σημαίνει ότι περιορίζεται η αναμονή για ανάκτηση κάποιου lock (όμως μπορεί κάποια διαδικασία add-remove να αποτύχει παραπάνω από μία φορά). Η Lazy synchronization έκδοση επίσης δε χρησιμοποιεί κλειδώματα στη συνάρτηση contains, ωστόσο για την εκτέλεση των add και remove απαιτείται local locking. Αυτό δικαιολογεί τη μείωση του throughput με την αύξηση των νημάτων πάνω από 8. Και στις δύο περιπτώσεις η βελτίωση της επίδοσης οφείλεται στην αποφυγή χρήσης lock μέσα στη συνάρτηση contains. Η συγκεκριμένη συνάρτηση καλείται αρκετές φορές στο πρόγραμμα, επομένως μειώνεται αισθητά ο χρόνος αναμονής για ανάκτηση κάποιου κλειδώματος (αυτός είναι ο κύριος λόγος, για τον οποίο οι δύο εκδόσεις παρουσιάζουν μεγαλύτερο throughput σε σχέση με τις Optimistic synchronization και Fine-grain locking).
- Επίσης η επίδοση της Optimistic synchronization υλοποίησης φαίνεται να είναι καλύτερη της



Εικόνα 5: Throuput για μέγεθος λίστας 1024.



Εικόνα 6: Throuput για μέγεθος λίστας 8192.

Fine-grain locking. Οι μέθοδοι add και remove στη δεύτερη περίπτωση υλοποιούνται μέσω hand-over-hand locking. Αυτό σημαίνει ότι απαιτούνται αρκετές λήψεις και απελευθερώσεις των κλειδωμάτων της δομής, ενώ μπορεί η ανάκτηση ενός lock από κάποια διεργασία-νήμα να παρεμποδίζει την προσπέλαση των στοιχείων της λίστας από τις υπόλοιπες διεργασίες. Είναι αναμενόμενο, λοιπόν, με την αύξηση των threads να αυξάνεται η αναμονή της ανάκτησης κάποιου lock (μείωση throughput). Στην περίπτωση της Optimistic synchronization έκδοσης, η διάσχιση της λίστας στις μεθόδους add και remove γίνεται χωρίς κλειδώματα. Αν το ζητούμενο στοιχείο βρεθεί, χρησιμοποιείται local locking για να γίνει έλεγχος της συνέπειας της δομής (εξετάζεται ότι δεν έχει αφαιρεθεί/προστεθεί κάποιο επιπλέον δεδομένο στη λίστα). Η διαφορά της επίδοσης των δύο υλοποιήσεων οφείλεται ακριβώς στο γεγονός ότι για την προσθήκη-/αφαίρεση ενός στοιχείου, η διάσχιση της λίστας στη μία περίπτωση δεν απαιτεί ανάκτηση κάποιου κλειδώματος.

- Τέλος παρατηρείται ότι ο συνδυασμός των λειτουργιών επηρεάζει άμεσα την επίδοση της εκτέλεσης του προγράμματος. Συγκεκριμένα ο συνδυασμός 80/10/10 παρουσιάζει μεγαλύτερο throughput από τον 20/40/40. Αυτό οφείλεται στον τρόπο υλοποίησης των μεθόδων add-

remove-contains, καθώς ορισμένες μέθοδοι μπορεί να επιβαρύνουν την επίδοση σε μεγαλύτερο βαθμό από τις υπόλοιπες μεθόδους (αυξάνοντας την αναμονή ανάκτησης κάποιου lock). Για παράδειγμα για την μέθοδο contains αναλόγως την υλοποίηση, μπορεί να χρησιμοποιούνται ή όχι κλειδώματα με αποτέλεσμα η αύξηση/μείωση του ποσοστού της λειτουργίας contains να οδηγεί σε βελτίωση ή όχι του throughput.

Κώδικας

Παρακάτω φαίνονται τα αρχεία που τροποποιήθηκαν για τα προηγούμενα ερωτήματα.

accounts.c

```
/**
 * A very simple parallel application
 * that handles an array of bank accounts.
 **/
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

#include "../common/timer.h"
#include "../common/aff.h"

#define MAX_THREADS 64
#define RUNTIME 10

#define print_error_and_exit(format...) \
    do { \
        fprintf(stderr, format); \
        exit(EXIT_FAILURE); \
    } while (0)

/**
 * Global data.
**/
pthread_barrier_t start_barrier;
int time_to_leave;

/**
 * The accounts' array.
 **/
struct {
    unsigned int value;
    char padding[64 - 2*sizeof(unsigned int)];
} accounts[MAX_THREADS];

/**
 * The struct that is passed as an argument to each thread.
```

```
**/
typedef struct {
    int tid,
        cpu;
    unsigned long long ops;

    /* Why do we use padding here? */
    char padding[64 - 2*sizeof(int) - sizeof(unsigned long long)];
} tdata_t;

void *thread_fn(void *targ);

int main()
{
    timer_tt *wall_timer;
    pthread_t threads[MAX_THREADS];
    tdata_t threads_data[MAX_THREADS];
    unsigned int nthreads = 0, *cpus;
    unsigned int i;

    //> Initializations.
    get_mtconf_options(&nthreads, &cpus);
    mt_conf_print(nthreads, cpus);
    if (pthread_barrier_init(&start_barrier, NULL, nthreads+1))
        print_error_and_exit("Failed to initialize start_barrier.\n"
           );
    wall_timer = timer_init();

    //> Spawn threads.
    for (i=0; i < nthreads; i++) {
        threads_data[i].tid = i;
        threads_data[i].cpu = cpus[i];
        threads_data[i].ops = 0;
        if (pthread_create(&threads[i], NULL, thread_fn, &
           threads_data[i]))
            print_error_and_exit("Error creating thread %d.\n", i);
    }

    //> Signal threads to start computation.
    pthread_barrier_wait(&start_barrier);
    timer_start(wall_timer);

    sleep(RUNTIME);
    time_to_leave = 1;

    //> Wait for threads to complete their execution.
    for (i=0; i < nthreads; i++) {
        if (pthread_join(threads[i], NULL))
            print_error_and_exit("Failure on pthread_join for thread
```

```
                ␣%d.\n", i);
    }

    timer_stop(wall_timer);

    //> How many operations have been performed by all threads?
    unsigned long long total_ops = 0;
    for (i=0; i < nthreads; i++)
        total_ops += threads_data[i].ops;

    //> Print results.
    double secs = timer_report_sec(wall_timer);
    double throughout = (double)total_ops / secs / 1000000.0;
    printf("Nthreads:␣%d␣␣Runtime(sec):␣%d␣␣Throughput(Mops/sec):␣
       %5.2lf\n",
            nthreads, RUNTIME, throughout);

    return EXIT_SUCCESS;
}

/**
 * This is the function that is executed by each spawned thread.
 **/
void *thread_fn(void *targ)
{
    tdata_t *mydata = targ;
    int i;

    //> Pin thread to the specified cpu.
    setaffinity_oncpu(mydata->cpu);

    //> Wait until master gives the green light!
    pthread_barrier_wait(&start_barrier);

    while (!time_to_leave) {
        for (i=0; i < 1000; i++)
            accounts[mydata->tid].value++;
        mydata->ops += 1000;
    }

    return NULL;
}
```

pthread_lock.c

```
#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);
    if ( pthread_spin_init(&lock->spinlock, PTHREAD_PROCESS_SHARED)
       ) {
        perror("Spinlock initialization");
        exit(1);
    }
    return lock;
}

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

void lock_acquire(lock_t *lock)
{
    if ( pthread_spin_lock(&lock->spinlock) ) {
        perror("Spinlock acquire");
        exit(1);
    }
}

void lock_release(lock_t *lock)
{
    if ( pthread_spin_unlock(&lock->spinlock) ) {
        perror("Spinlock release");
        exit(1);
    }
}
```

ttas_lock.c

```
#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)
{
    while (1) {
        while (lock->state) ;
        if (__sync_lock_test_and_set(&lock->state, LOCKED) ==
           UNLOCKED)
            return;
    }
}

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

array_lock.c

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

__thread int mySlotIndex;

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


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

    XMALLOC(lock, 1);
```

```
    XMALLOC(lock->flag, nthreads);
    lock->size = nthreads;
    lock->flag[0] = 1;
    lock->tail = 0;

    return lock;
}

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

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

}

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

ll_fgl.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 node_lock;
} ll_node_t;

struct linked_list {
    ll_node_t *head;
};

/**
 * Functions to acquire and release spinlock
```

```
 **/
void lock_acquire(pthread_spinlock_t *lock)
{
    if ( pthread_spin_lock(lock) ) {
        perror("Spinlock acquire");
        exit(1);
    }
}
void lock_release(pthread_spinlock_t *lock)
{
    if ( pthread_spin_unlock(lock) ) {
        perror("Spinlock release");
        exit(1);
    }
}

/**
 * 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;
    if ( pthread_spin_init(&ret->node_lock, PTHREAD_PROCESS_SHARED)
       ) {
        perror("Spinlock initialization");
        exit(1);
    }

    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);
}

int ll_contains(ll_t *ll, int key)
{
    ll_node_t *curr = ll->head;
    int ret = 0;

    /* ll_contains doesn't need locks */
    while (curr->key < key)
        curr = curr->next;

    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;

    curr = ll->head;
    lock_acquire(&curr->node_lock);
    next = curr->next;
    lock_acquire(&next->node_lock);

    while (next->key < key) {
        lock_release(&curr->node_lock);
```

```
        curr = next;
        next = curr->next;
        lock_acquire(&next->node_lock);
    }

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

    lock_release(&next->node_lock);
    lock_release(&curr->node_lock);

    return ret;
}

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

    curr = ll->head;
    lock_acquire(&curr->node_lock);
    next = curr->next;
    lock_acquire(&next->node_lock);

    while (next->key < key) {
        lock_release(&curr->node_lock);
        curr = next;
        next = curr->next;
        lock_acquire(&next->node_lock);
    }

    if (key == next->key) {
        ret = 1;
        curr->next = next->next;
        /* In this implementation we can delete the nodes as soon as
            we remove them
         * but it is not that simple in optimistic, lazy and non-
            blocking implementations.
         * So we won't do it here to have comparable time results.
         */
        // ll_node_free(next);
    }

    lock_release(&next->node_lock);
    lock_release(&curr->node_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");
}
```

ll_opt.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 node_lock;
} ll_node_t;

struct linked_list {
    ll_node_t *head;
};

/**
 * Functions to acquire and release spinlock
 **/
void lock_acquire(pthread_spinlock_t *lock)
{
    if ( pthread_spin_lock(lock) ) {
        perror("Spinlock␣acquire");
        exit(1);
    }
```

```
}
void lock_release(pthread_spinlock_t *lock)
{
    if ( pthread_spin_unlock(lock) ) {
        perror("Spinlock release");
        exit(1);
    }
}

/**
 * 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;
    if ( pthread_spin_init(&ret->node_lock, PTHREAD_PROCESS_SHARED)
       ) {
        perror("Spinlock initialization");
        exit(1);
    }

    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);
}

/**
 * Validate consistency of list
 **/
int validate(ll_t *ll, ll_node_t *curr, ll_node_t *next)
{
    int key = curr->key;
    ll_node_t *temp = ll->head;

    while (temp->key <= key) {
        if (temp == curr)
            return curr->next == next;
        temp = temp->next;
    }

    return 0;
}

int ll_contains(ll_t *ll, int key)
{
    ll_node_t *curr = ll->head;
    int ret = 0;

    /* ll_contains doesn't need locks */
    while (curr->key < key)
        curr = curr->next;

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

int ll_add(ll_t *ll, int key)
{
    int ret = 0, br = 1;
```

```
    ll_node_t *curr, *next;
    ll_node_t *new_node;

    while (br) {
        curr = ll->head;
        next = curr->next;

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

        lock_acquire(&curr->node_lock);
        lock_acquire(&next->node_lock);

        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;
            }
            br = 0;
        }

        lock_release(&curr->node_lock);
        lock_release(&next->node_lock);
    }

    return ret;
}

int ll_remove(ll_t *ll, int key)
{
    int ret = 0, br = 1;
    ll_node_t *curr, *next;

    while (br) {
        curr = ll->head;
        next = curr->next;

        while (next->key < key) {
            // if (next->key == key)
            //  break;
            curr = next;
            next = curr->next;
        }

        lock_acquire(&curr->node_lock);
```

```
        lock_acquire(&next->node_lock);

        if ( validate(ll, curr, next) ) {
            if (key == next->key) {
                ret = 1;
                curr->next = next->next;
            }
            br = 0;
        }

        lock_release(&curr->node_lock);
        lock_release(&next->node_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");
}
```

ll_lazy.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 node_lock;
    int valid;
} ll_node_t;
```

```
struct linked_list {
    ll_node_t *head;
};

/**
 * Functions to acquire and release spinlock
 **/
void lock_acquire(pthread_spinlock_t *lock)
{
    if ( pthread_spin_lock(lock) ) {
        perror("Spinlock acquire");
        exit(1);
    }
}
void lock_release(pthread_spinlock_t *lock)
{
    if ( pthread_spin_unlock(lock) ) {
        perror("Spinlock release");
        exit(1);
    }
}

/**
 * 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;
    if ( pthread_spin_init(&ret->node_lock, PTHREAD_PROCESS_SHARED)
       ) {
        perror("Spinlock initialization");
        exit(1);
    }
    ret->valid = 1;

    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);
}

/**
 * Validate consistency of list
 **/
int validate(ll_node_t *curr, ll_node_t *next)
{
    return curr->valid
            && next->valid
            && curr->next == next;
}

int ll_contains(ll_t *ll, int key)
{
    ll_node_t *curr = ll->head;
    int ret = 0;

    /* ll_contains doesn't need locks */
    while (curr->key < key)
        curr = curr->next;
```

```
    ret = (key == curr->key) && curr->valid;
    return ret;
}

int ll_add(ll_t *ll, int key)
{
    int ret = 0, br = 1;
    ll_node_t *curr, *next;
    ll_node_t *new_node;

    while (br) {
        curr = ll->head;
        next = curr->next;

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

        lock_acquire(&curr->node_lock);
        lock_acquire(&next->node_lock);

        if ( validate(curr, next) ) {
            if (key != next->key) {
                ret = 1;
                new_node = ll_node_new(key);
                new_node->next = next;
                curr->next = new_node;
            }
            br = 0;
        }

        lock_release(&curr->node_lock);
        lock_release(&next->node_lock);
    }

    return ret;
}

int ll_remove(ll_t *ll, int key)
{
    int ret = 0, br = 1;
    ll_node_t *curr, *next;

    while (br) {
        curr = ll->head;
        next = curr->next;
```

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

        lock_acquire(&curr->node_lock);
        lock_acquire(&next->node_lock);

        if ( validate(curr, next) ) {
            if (key == next->key) {
                next->valid = 0;
                curr->next = next->next;
                ret = 1;
            }
            br = 0;
        }

        lock_release(&curr->node_lock);
        lock_release(&next->node_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");
}
```

ll_nb.c

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

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

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

struct linked_list {
    ll_node_t *head;
};

/**
 * Create a new linked list node.
 **/
static ll_node_t *ll_node_new(int key)
{
    ll_node_t *ret;

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

    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 *get(ll_node_t *refer, unsigned *marked)
{
    long addr;

    addr = (long) refer;
    *marked = addr & 1;

    return (ll_node_t *) (addr & ~1);
}

void find(ll_node_t **bef, ll_node_t **aft, ll_t *ll, int key)
{
    ll_node_t *curr, *next, *succ;
    int snip;
    unsigned marked;
retry:
    while (1) {
        curr = ll->head;
        next = get(curr->next, &marked);

        while (1) {
            succ = get(next->next, &marked);
            while (marked) {
                snip = __sync_bool_compare_and_swap(&curr->next,
                   next, succ);
                if (!snip)
                    goto retry;
                next = succ;
                succ = get(next->next, &marked);
            }

            if (next->key >= key) {
                *bef = curr;
                *aft = next;
                return;
            }
            curr = next;
```

```
            next = succ;
        }
    }
}

int ll_contains(ll_t *ll, int key)
{
    ll_node_t *curr = ll->head;
    int ret = 0;
    unsigned marked;

    /* ll_contains doesn't need locks */
    while (curr->key < key) {
        curr = get(curr->next, &marked);
        // Could avoid get but whatever
    }

    ret = (key == curr->key && !marked);
    return ret;
}

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

    while (1) {
        find(&curr, &next ,ll, key);
        if (next->key == key) {
            ll_node_free(new_node);
            return 0;
        }
        else {
            new_node->next = next; //marked field will be zero this
               way
            if ( __sync_bool_compare_and_swap(&curr->next, next,
               new_node) )
                return 1;
        }
    }
}

int ll_remove(ll_t *ll, int key)
{
    int snip;
    unsigned marked;
    ll_node_t *curr, *next, *succ, *dirty;

    while (1) {
```

```
        find(&curr, &next, ll, key);
        if (next->key != key) {
            return 0;
        }
        else {
            succ = get(next->next, &marked);
            dirty = (ll_node_t *) ( (uintptr_t) succ | 1 );
            snip = __sync_bool_compare_and_swap(&next->next, succ,
               dirty);
            if (!snip) continue;
            __sync_bool_compare_and_swap(&curr->next, next, succ);
            return 1;
        }
    }
}

/**
 * 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");
}
```