Τελική Αναφορά 3ης Άσκησης

Βιτάλιος Σαλής - 03115751
Φίλιππος Μαλανδράκης - 03116200

Ζήτημα 1 - Λογαριασμοί Τράπεζας

Εισαγωγή

Σε αυτό το ζήτημα μας δίνεται παραλληλοποιημένος κώδικας για την ταυτόχρονη πρόσβαση τραπεζικών λογαριασμών από διάφορα threads, και καλούμαστε να:

1) Λάβουμε μετρήσεις του throughput για 2 διαφορετικά σενάρια χαρτογραφήσεων νημάτων σε πυρήνες και να τις επιδείξουμε σε γραφήματα.
2) Επιχειρήσουμε να βελτιώσουμε το πρόγραμμα που μας δίνεται, επαναλαμβάνοντας στη συνέχεια το παραπάνω βήμα.

Προσδοκίες - Εκτέλεση μετρήσεων

Αρχικά, παρατηρούμε πως αν και δεν έχουμε ταυτόχρονη πρόσβαση σε κοινά δεδομένα(κάθε thread επικεντρώνεται σε ένα κελί του πίνακα, δηλαδή ένα λογαριασμό), επιθυμούμε η δραστηριοποίηση όλων των threads να ξεκινήσει και να τελειώσει την ίδια στιγμή. Ο συγχρονισμός αυτός επιτυγχάνεται με τη χρήση barriers.
Προφανώς, για κάθε διπλασιασμό των threads αναμένουμε να διπλασιάζεται και το throughput της εφαρμογής.
Για την παραγωγή του εκτελέσιμου accounts, κάνουμε make στην ουρά parlab το σχετικό Makefile, τρέχοντας το ακόλουθο script make_on_queue.sh :

```
#!/bin/bash

## Give the Job a descriptive name
#PBS -N make_accounts

## Output and error files
#PBS -o make_accounts.out
#PBS -e make_accounts.err
```

```
## How many machines should we get?
#PBS -l nodes=1:ppn=1

##How long should the job run for?
#PBS -l walltime=00:10:00

## Start
## Run make in the src folder (modify properly)

cd /home/parallel/parlab13/a3/z1
make
```

Για να εκτελέσουμε τώρα το πρώτο σενάριο μετρήσεων, τρέχουμε το ακόλουθο script run_on_queue.sh στην ουρά του sandman:

```
#!/bin/bash

## Give the Job a descriptive name
#PBS -N run_accounts1

## Output and error files
#PBS -o run_accounts1.out
#PBS -e run_accounts1.err

##How long should the job run for?
#PBS -l walltime=00:10:00

## Start
## Run make in the src folder (modify properly)

cd /home/parallel/parlab13/a3/z1
MT_CONF=0 ./accounts
MT_CONF=0,1 ./accounts
MT_CONF=0,1,2,3 ./accounts
MT_CONF=0,1,2,3,4,5,6,7 ./accounts
MT_CONF=0,1,2,3,4,5,6,7,32,33,34,35,36,37,38,39 ./accounts
MT_CONF=0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47 ./accounts
MT_CONF=0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63 ./accounts
```

Με παρόμοιο τρόπο τρέχουμε και τις μετρήσεις για το 2ο σενάριο εκτέλεσης.

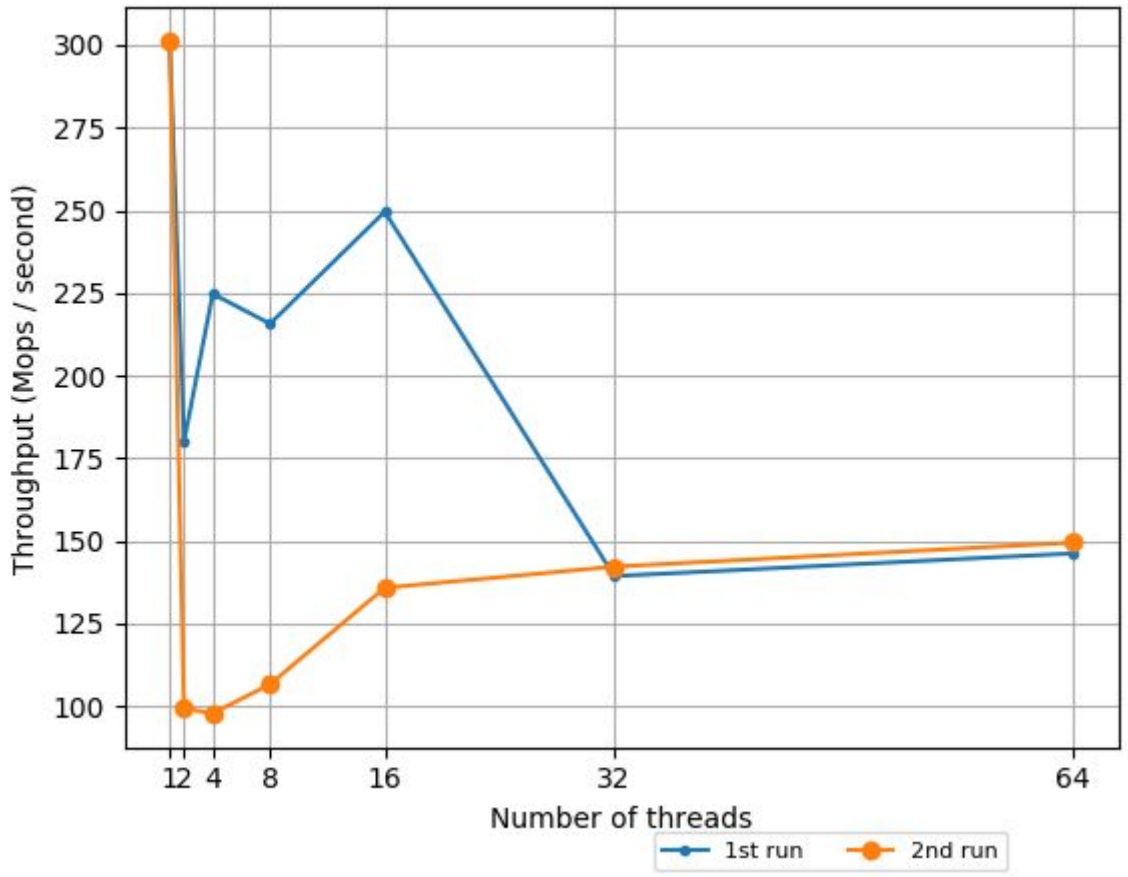
Επίδειξη Μετρήσεων

Για να δημιουργησουμε γραφήματα, υλοποιήσαμε το ακόλουθο Python πρόγραμμα:

```
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import numpy as np
import sys

def parse_file(fname):
        with open(fname) as f:
                lines = f.readlines()

        cnt = 1
        size_stats = {}
        for line in lines:
            if line.startswith("Nthreads:"):
                # line of the form:
                # Nthreads: N Runtime(sec): 10 Throughput(Mops/sec): t
                splitted = line.split()
                # run
                if cnt < 8:
                    run = 1
                else:
                    run = 2
                # throughput
                throughput = splitted[5].strip()
                # thread number
                thread_num = splitted[1].strip()
                if not size_stats.get(run, None):
                    size_stats[run] = []

                size_stats[run].append({"throughput": throughput, "nthread":
thread_num})
                cnt += 1
        return size_stats

if len(sys.argv) < 2:
    print ("Usage plot_metrics.py <input_file>")
    exit(-1)

stats_by_size = parse_file(sys.argv[1])
markers = ['.', 'o', 'v', '*', 'D', 'X']
```

```
x_ticks = [1, 2, 4, 8, 16, 32, 64]
fig = plt.figure(1)
plt.grid(True)
ax = plt.subplot(111)
ax.set_xlabel("Number of threads")
ax.set_ylabel("Throughput (Mops / second)")
ax.xaxis.set_ticks(x_ticks)
ax.xaxis.set_ticklabels(map(str, x_ticks))

for j, size in enumerate(sorted(stats_by_size.keys(), key=lambda x: int(x))):
    stats = stats_by_size[size]
    y_axis = [0 for _ in range(len(x_ticks))]
    for stat in stats:
        pos = x_ticks.index(int(stat["nthread"]))

        y_axis[pos] = float(stat["throughput"])
    if size == 1:
        ax.plot(x_ticks, tuple(y_axis), label="1st run", marker=markers[j])
    else:
        ax.plot(x_ticks, tuple(y_axis), label="2nd run", marker=markers[j])

lgd = ax.legend(ncol=len(stats_by_size.keys()), bbox_to_anchor=(0.9, -0.1),
prop={'size':8})
plt.savefig("accounts-initial.png", bbox_extra_artists=(lgd,),
bbox_inches='tight')
```

Ακολουθεί το γράφημα για τα throughput των 2 σεναρίων εκτέλεσης.



Σχολιασμός Αποτελεσμάτων

Σε πρώτη φάση, αναφέρουμε ότι τα αποτελέσματα είναι απογοητευτικά, καθώς και στις 2 περιπτώσεις η αύξηση των πυρήνων αντί να βελτιώσει το throughput, το επιβαρύνει!
Συγκεκριμένα, στο πρώτο σενάριο η όποια “βελτίωση” σημειώνεται μεταξύ 2-16 threads, όπου χρησιμοποιούμε μόνο έναν CPU(λόγω των πυρήνων που έχουν επιλεγεί). Για 32 και 64 threads, με την εισαγωγή των υπόλοιπων 3 CPUs το throughput καταρρακώνεται περαιτέρω. Αυτό πιθανότατα αποδίδεται στο γεγονός ότι ακολουθούν πολιτική NUMA, πράγμα που προσθέτει overhead τόσο στην πρόσβαση των πιο απομακρυσμένων RAM(3 στις 4), όσο και στο συγχρονισμό των threads.
Στο δεύτερο σενάριο, από τα 4 μόλις threads έχουν ενεργοποιηθεί και οι 4 CPUs, με τα αίτια που αναπτύχθηκαν και παραπάνω να κρατάνε το throughput σε πολύ χαμηλά στάνταρ. Με τη σταδιακή ενεργοποίηση και των υπόλοιπων threads των CPUs έχουμε και πάλι μια αμελητέα

βελτίωση μπροστά στην θεωρητικά αναμενόμενη.
Η αποτυχία της παραλληλοποίησης της εφαρμογής οφείλεται στο γεγονός ότι κάθε thread, όταν πάει να φορτώσει στην cache το στοιχείο που θα επεξεργαστεί, φορτώνει και άλλα στοιχεία του πίνακα(συνολικα 16, αφού έχουμε block size 64 και κάθε int κοστίζει 4B). Αν λοιπόν κάποιο από τα υπόλοιπα στοιχεία έχει υποστεί μεταβολή(πράγμα που συμβαίνει διαρκώς), θα πρέπει το thread να ενημερωθεί πρώτα για τις νέες τιμές και μετά να προχωρήσει στην επεξεργασία του στοιχείου του. Πρέπει λοιπόν να βρούμε κάποιον τρόπο ώστε το κάθε thread να φορτώνει μόνο το στοιχείο που πραγματικά χρειάζεται, αγνοώντας τις μεταβολές των υπολοίπων στοιχείων.

Βελτιωμένη Υλοποίηση

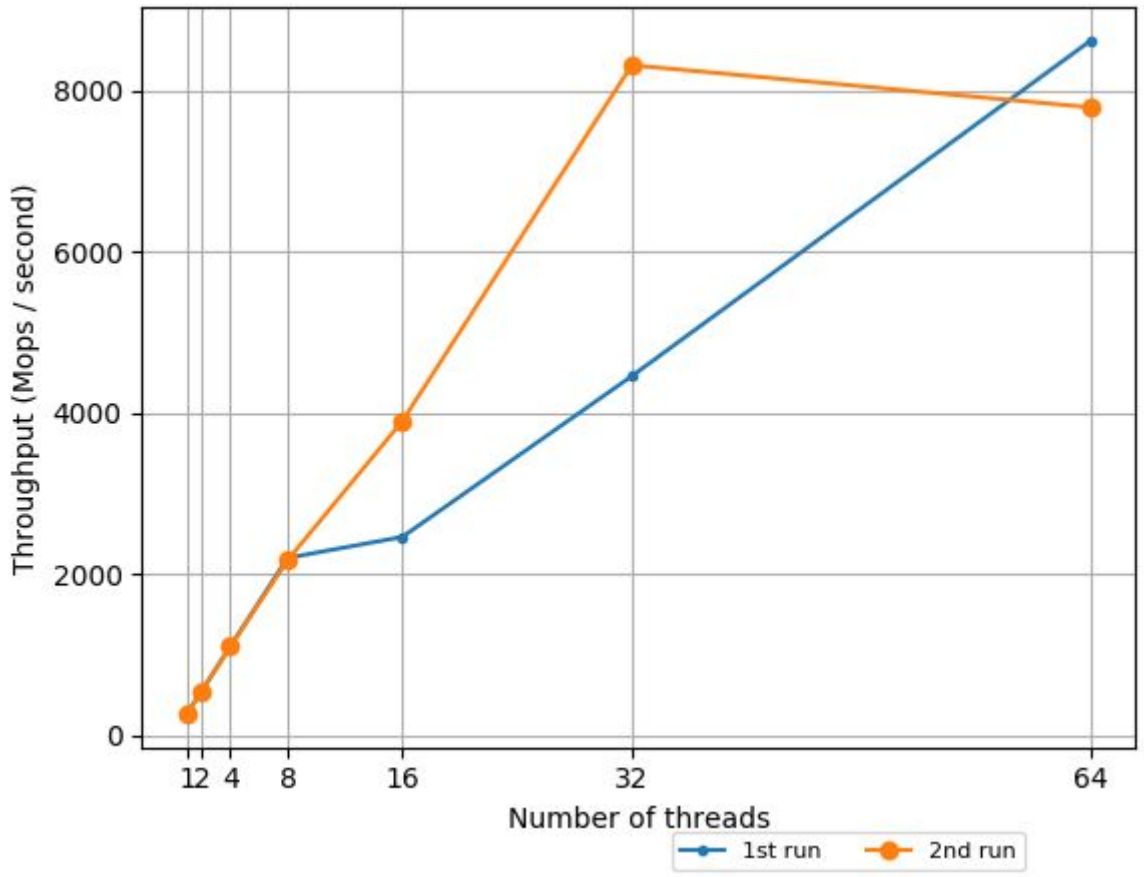
Για να επιτύχουμε το παραπάνω, εισάγουμε padding μεγέθους 60B σε κάθε στοιχείο του πίνακα των accounts. Άρα πλέον η εικόνα του struct accounts είναι η εξης:

```
struct {
    unsigned int value;

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

Η τεχνική αυτή χρησιμοποιείται αντίστοιχα και στο struct που περιέχει τα data του κάθε thread, το οποίο και δε χρειάζεται να πειράξουμε.

Επίδειξη Μετρήσεων

Μεταγλωττίζοντας και τρέχοντας εκ νέου το πρόγραμμα στην ουρά sandman, λάβαμε τις ανανεωμένες μετρικές για τα throughput. Στη συνέχεια παρουσιάζουμε το νέο γράφημα για τα throughput των 2 σεναρίων, που παράχθηκε με το ίδιο Python script με προηγουμένως.



Συμπεράσματα

Εδώ πλέον και στα 2 runs έχουμε σχεδόν ιδεατά αποτελέσματα, δηλαδή διπλασιασμό throughput για κάθε διπλασιασμό threads, με 2 εξαιρέσεις. Στο 1ο σενάριο, από τα 8 στα 16 threads και στο 2ο σενάριο, από τα 32 στα 64 threads δεν παρατηρείται η αναμενόμενη κλιμάκωση. Και στις 2 περιπτώσεις αυτό που οφείλεται είναι η ενεργοποίηση όλων των threads ενός CPU(ενώ πριν ήταν ενεργοποιημένα τα μισά). Έτσι, πλέον για κάθε πυρήνα(και L1 cache) ανταγωνίζονται 2 threads, προσθέτοντας σημαντικό overhead στην υλοποίησή μας.

Ζήτημα 2 - Αμοιβαίος Αποκλεισμός - Κλειδώματα

Εισαγωγή

Σε αυτό το ζήτημα μας δίνεται παραλληλοποιημένος κώδικας για την ταυτόχρονη πρόσβαση σε μια συνδεδεμένη λίστα από διάφορα threads, καθώς και μερικά είδη κλειδωμάτων. Εμείς καλούμαστε να:

1) Υλοποιήσουμε όσα κλειδώματα δεν είναι ολοκληρωμένα.
2) Εξετάσουμε την απόδοση των κλειδωμάτων για διάφορους συνδυασμούς threads - μεγεθών λίστας.

Υλοποίηση

Παρακάτω θα επιδείξουμε τα locks που υλοποιήσαμε. Ξεκινάμε με το **ttas_lock**, η λογική του οποίου είναι παρεμφερής με του **tas_lock**, με τη διαφορά ότι πρώτα περιμένει να απελευθερωθεί το κλείδωμα και μετά τρέχει τη συνάρτηση testAndSet().

```
#include <stdbool.h>

#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(true) {
        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);
}
```

Όπως προαναφέρθηκε, λοιπόν, η μόνη διαφοροποίηση από το **tas_lock** είναι στη συνάρτηση *lock_acquire*, όπου πρώτα αναμένουμε την απελευθέρωση του λουκέτου και μετά προσπαθούμε να το κλειδώσουμε. Αν ωστόσο προλάβει άλλος, επαναλαμβάνουμε τη διαδικασία.

Συνεχίζουμε με το **pthread_lock**, το οποίο βασίζεται στη βιβλιοθήκη Pthreads και συγκεκριμένα στο κλείδωμα *pthread_spinlock_t*.

```
#include <pthread.h>

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

```c
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_SHARED);
    return lock;
}

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

void lock_acquire(lock_t *lock)
{
    while(pthread_spin_trylock(&lock->spinlock) != 0);
}

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

Στη συνάρτηση *lock_init* αρχικοποιούμε το spinlock μας σε κατάσταση UNLOCKED, μέσω της συνάρτησης *pthread_spin_init*. Η *lock_acquire* προσπαθεί διαρκώς να “πάρει” το spinlock, πράγμα που επιτυγχάνει μόλις αυτό γίνει διαθέσιμο. Η *lock_release* απλώς ελευθερώνει το spinlock.

Τέλος, παρουσιάζουμε το **array_lock**.

```c
#include <stdbool.h>


#include "lock.h"
```

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

struct lock_struct {
    bool flags[MAX_THREADS];
    int tail;
    int nthreads;
};

__thread int mySlot;

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

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

void lock_acquire(lock_t *lock)
{
    mySlot = __atomic_fetch_add(&lock->tail, 1, __ATOMIC_RELAXED) %
lock->nthreads;
    while(lock->flags[mySlot] != true);
    lock->flags[mySlot] = false;
}

void lock_release(lock_t *lock)
{
    __atomic_store_n(&lock->flags[(mySlot + 1) % lock->nthreads], true,
__ATOMIC_RELEASE);
}
```

Εδώ, στο *lock_struct* αποθηκεύουμε τον πίνακα με τα flags των threads και τις μεταβλητές *tail, nthreads*. Στη *lock_init* αρχικοποιούμε κατάλληλα τις μεταβλητές που αναφέραμε, ενώ έχουμε ορίσει ακόμα και μια τοπική μεταβλητή ανα thread, *mySlot*, στην οποία σε κάθε στιγμή υπάρχει το τελευταίο index του πίνακα *flags* το οποίο αντιστοιχούσε στο thread. Η

lock_acquire υπολογίζει ατομικά τη νέα τιμή του *mySlot* και περιμένει έως ότου το predecessor thread δώσει το OK για να πάρει το lock. Τέλος, η *lock_release* ελευθερώνει ατομικά το lock.

Εκτέλεση μετρήσεων

Για την παραγωγή των εκτελέσιμων linked_list και test, κάνουμε make στην ουρά parlab το σχετικό Makefile(αλλάζοντας κατάλληλα κάθε φορά την παράμετρο *LOCK_FILE*), τρέχοντας το ακόλουθο script make_on_queue.sh :

```
#!/bin/bash

## Give the Job a descriptive name
#PBS -N make_locks

## Output and error files
#PBS -o make_locks.out
#PBS -e make_locks.err

## How many machines should we get?
#PBS -l nodes=1:ppn=1

##How long should the job run for?
#PBS -l walltime=00:10:00

## Start
## Run make in the src folder (modify properly)

cd /home/parallel/parlab13/a3/z2
make
```

Για να ελέγξουμε την ορθότητα των κλειδωμάτων, εκτελούμε το παρακάτω script test_on_queue.sh στην ουρά του sandman:

```
#!/bin/bash

## Give the Job a descriptive name
#PBS -N test_ttas

## Output and error files
#PBS -o test_ttas.out
#PBS -e test_ttas.err
```

```
##How long should the job run for?
#PBS -l walltime=00:01:00

## Start
## Run make in the src folder (modify properly)

cd /home/parallel/parlab13/a3/z2
nthreads=( 1 2 4 8 16 32 64 )

for i in "${nthreads[@]}"
do
    ./test ${i}
done
```

Για να εκτελέσουμε τώρα τις ζητούμενες μετρήσεις για κάθε κλείδωμα, τρέχουμε το ακόλουθο script run_on_queue.sh στην ουρά του sandman:

```
#!/bin/bash

## Give the Job a descriptive name
#PBS -N run_pthread

## Output and error files
#PBS -o run_pthread.out
#PBS -e run_pthread.err

##How long should the job run for?
#PBS -l walltime=00:10:00

## Start
## Run make in the src folder (modify properly)

cd /home/parallel/parlab13/a3/z2

sizes=( 16 1024 8192 )

for i in "${sizes[@]}"
do
    MT_CONF=0 ./linked_list ${i}
    MT_CONF=0,1 ./linked_list ${i}
    MT_CONF=0,1,2,3 ./linked_list ${i}
    MT_CONF=0,1,2,3,4,5,6,7 ./linked_list ${i}
    MT_CONF=0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 ./linked_list ${i}
MT_CONF=0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,2
```

```
7,28,29,30,31 ./linked_list ${i}
MT_CONF=0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,2
7,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,
54,55,56,57,58,59,60,61,62,63 ./linked_list ${i}
done
```

Επίδειξη Μετρήσεων

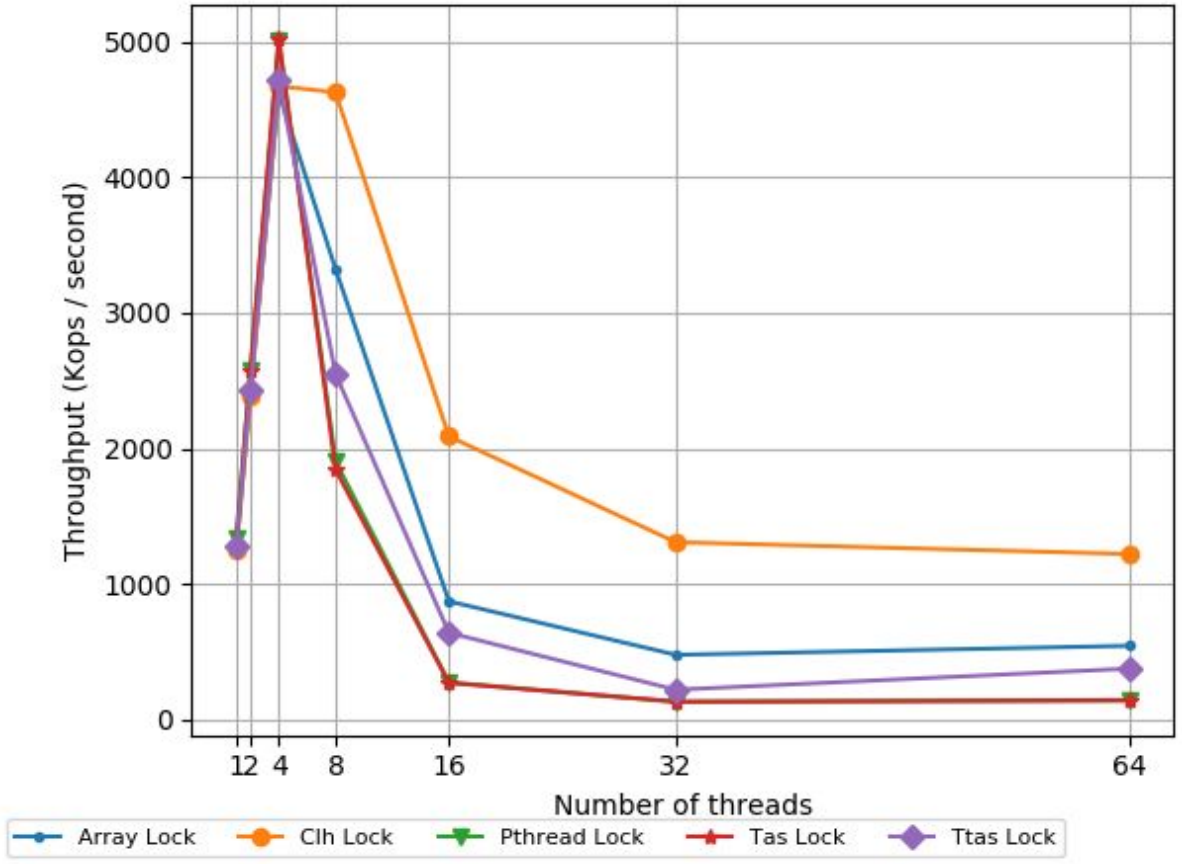
Για να δημιουργησουμε γραφήματα, υλοποιήσαμε το ακόλουθο Python πρόγραμμα:

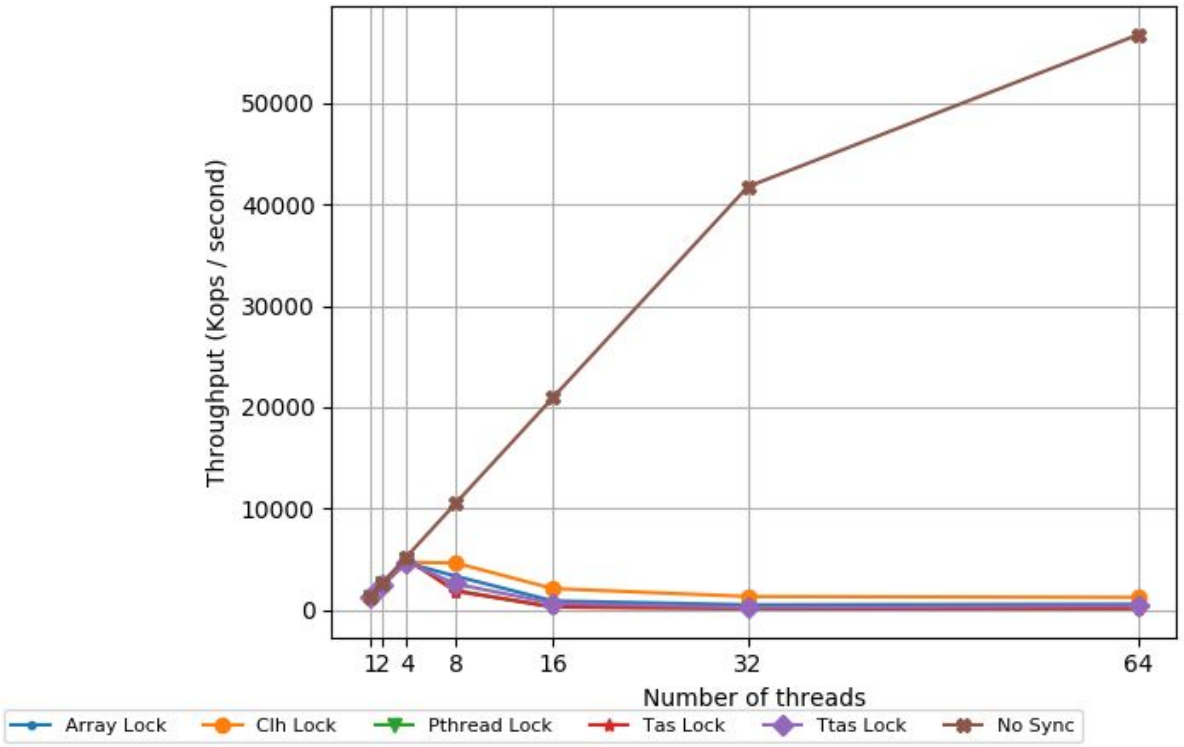
```python
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import numpy as np
import sys

def parse_file(fname):
        with open(fname) as f:
                lines = f.readlines()

        cnt = 1
        size_stats = {}
        for line in lines:
            if line.startswith("Nthreads:"):
                # line of the form:
                # Nthreads: N Runtime(sec): 10 Throughput(Mops/sec): t
                splitted = line.split()
                # lock
                if cnt > 7 and cnt < 15:
                    lock = "Tas Lock"
                elif cnt < 22:
                    lock = "Ttas Lock"
                elif cnt < 29:
                    lock = "Clh Lock"
                elif cnt < 36:
                    lock = "Pthread Lock"
                else:
                    lock = "Array Lock"
                # throughput
                throughput = splitted[5].strip()
                # thread number
                thread_num = splitted[1].strip()
                if not size_stats.get(lock, None):
                    size_stats[lock] = []
```

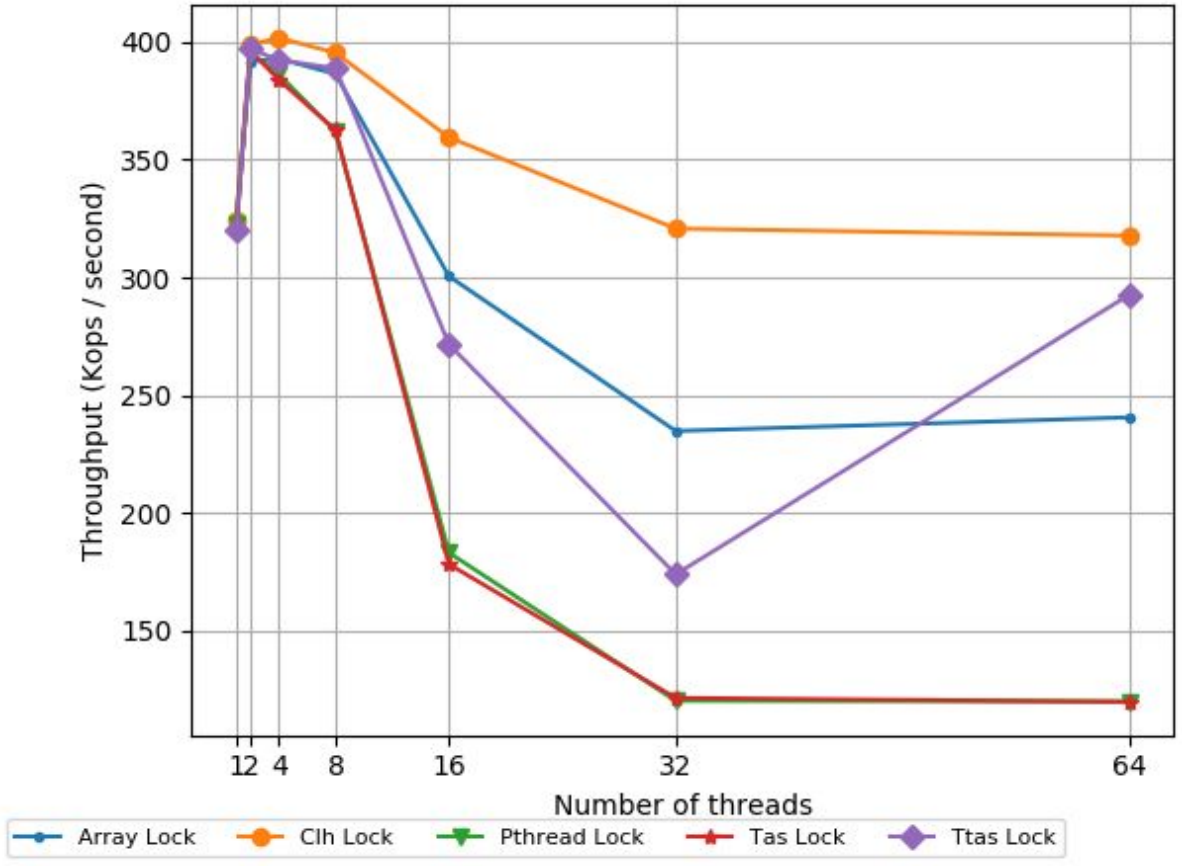
```
                size_stats[lock].append({"throughput": throughput, "nthread":
thread_num})
                cnt += 1
        return size_stats

if len(sys.argv) < 2:
    print ("Usage plot_metrics.py <input_file>")
    exit(-1)
stats_by_size = parse_file(sys.argv[1])
markers = ['.', 'o', 'v', '*', 'D', 'X']
print(stats_by_size)
x_ticks = [1, 2, 4, 8, 16, 32, 64]
fig = plt.figure(1)
plt.grid(True)
ax = plt.subplot(111)
ax.set_xlabel("Number of threads")
ax.set_ylabel("Throughput (Kops / second)")
ax.xaxis.set_ticks(x_ticks)
ax.xaxis.set_ticklabels(map(str, x_ticks))

for j, size in enumerate(sorted(stats_by_size.keys(), key=lambda x: sorted(x))):
    stats = stats_by_size[size]

    y_axis = [0 for _ in range(len(x_ticks))]
    for stat in stats:
        pos = x_ticks.index(int(stat["nthread"]))
        y_axis[pos] = float(stat["throughput"])
    ax.plot(x_ticks, tuple(y_axis), label=str(size), marker=markers[j])

lgd = ax.legend(ncol=len(stats_by_size.keys()), bbox_to_anchor=(0.9, -0.1),
prop={'size':8})
plt.savefig("linked_list-16.png", bbox_extra_artists=(lgd,),
bbox_inches='tight')
```

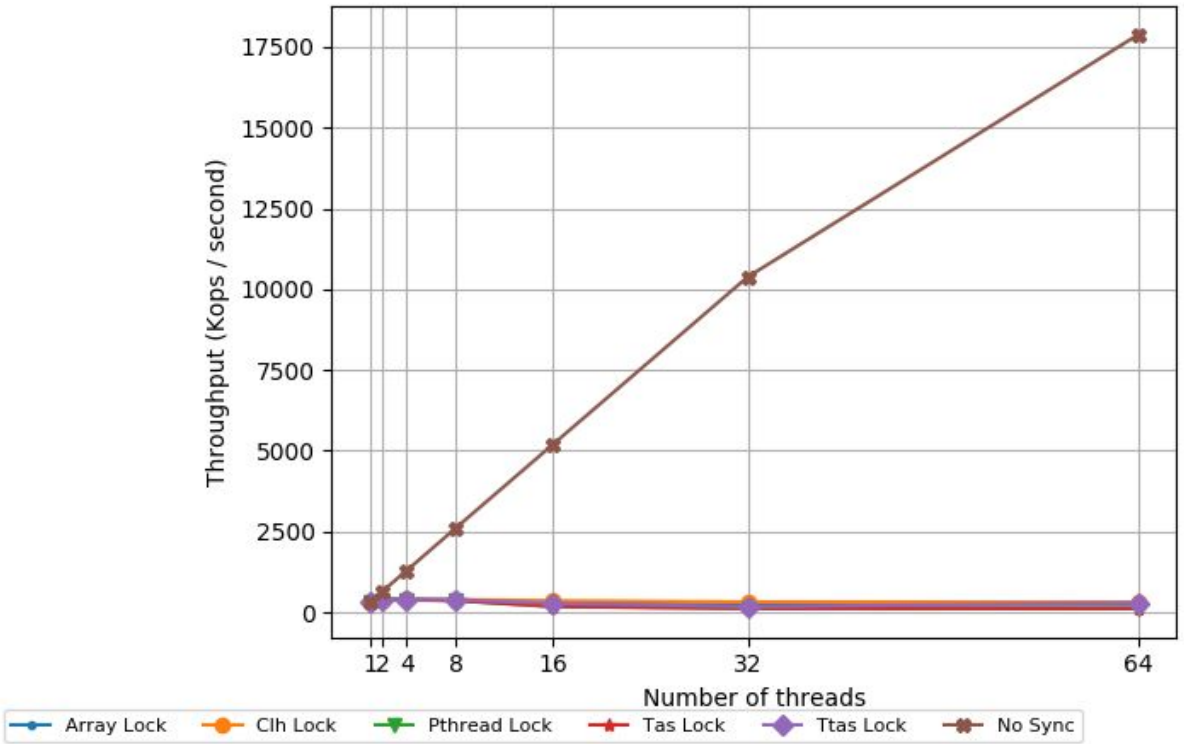
Υλοποιήσαμε ένα ακόμη ελάχιστα παραλλαγμένο script το οποίο εμπεριέχει και τις μετρήσεις που λήφθηκαν χωρίς κλείδωμα (**nosync_lock**), διαφέροντας χαοτικά από τις υπόλοιπες.

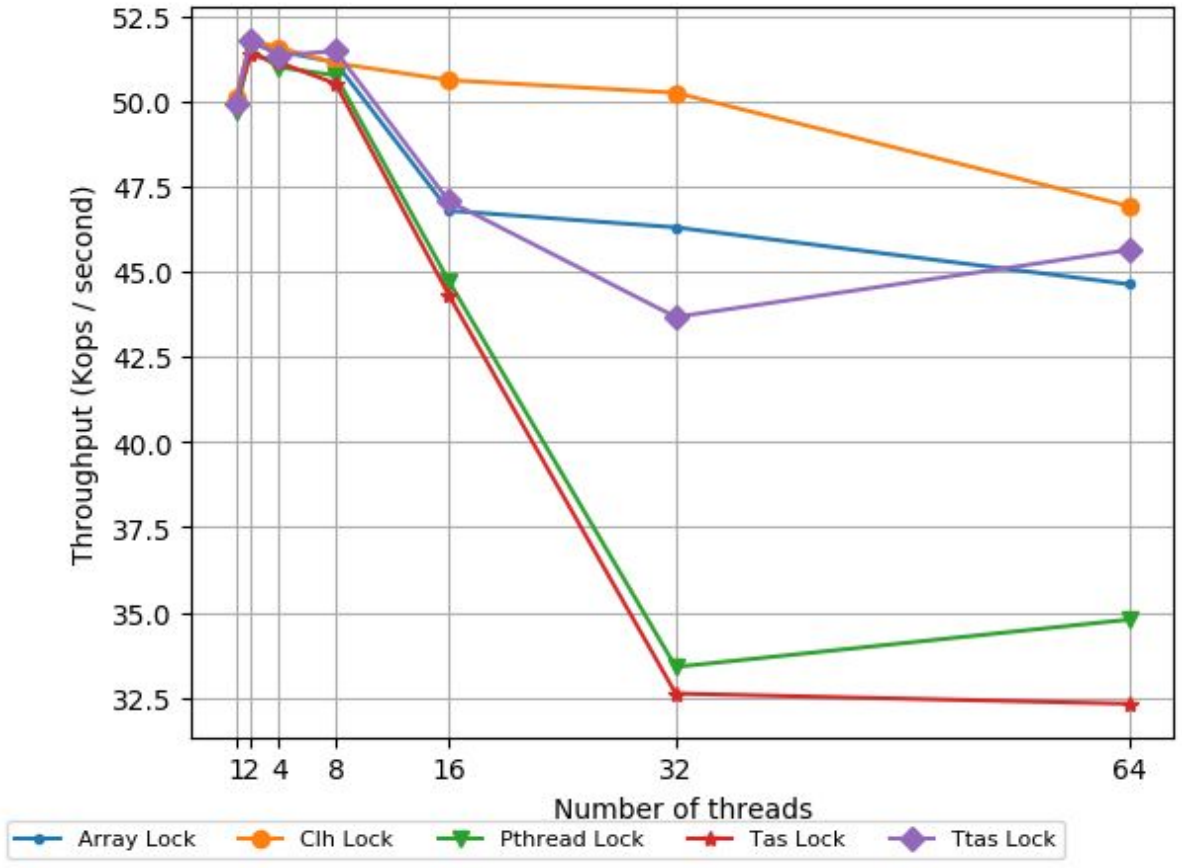
Παρακάτω, δίνουμε τα γραφήματα throughput που προέκυψαν με τη χρήση των διάφορων κλειδωμάτων, χωρίς και με το **nosync_lock**, για μεγέθη λίστας 16, 1024 και 8192 αντίστοιχα.

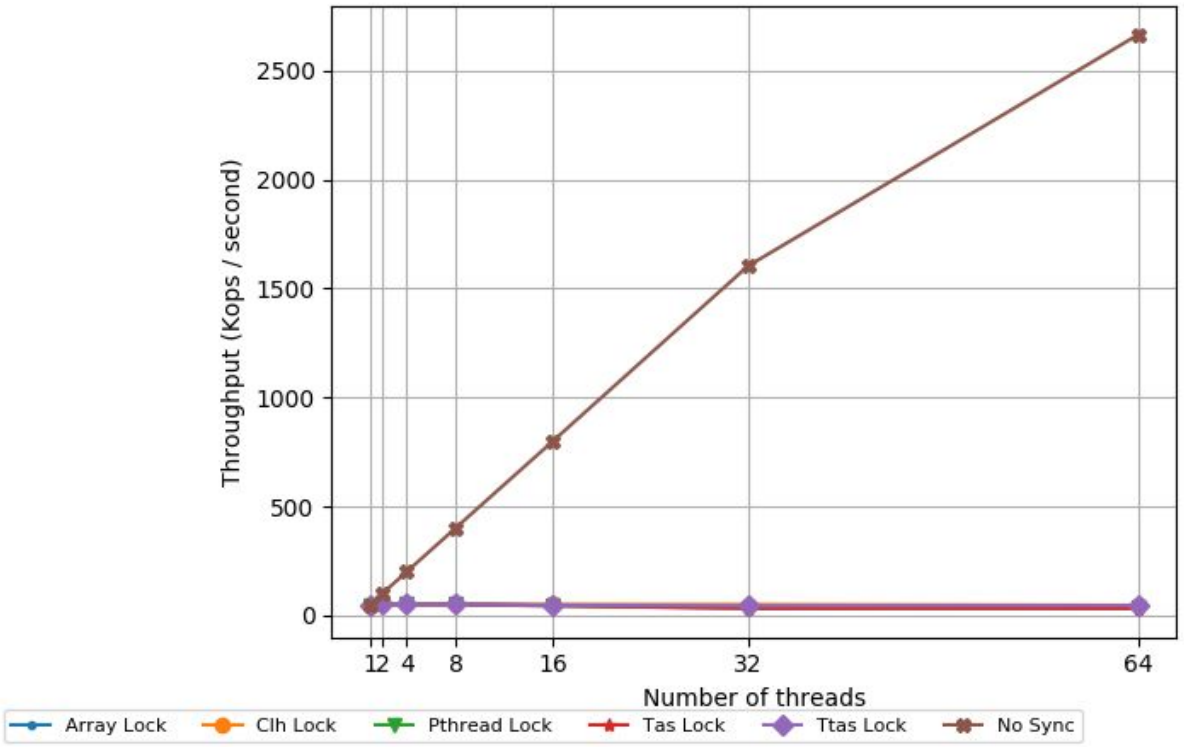

Throughput (Kops / second)
5000
4000
3000
2000
1000
0
1 2 4 8 16 32 64
Number of threads
Array Lock
Clh Lock
Pthread Lock
Tas Lock
Ttas Lock


Throughput (Kops / second)
50000
40000
30000
20000
10000
0
1 2 4 8 16 32 64
Number of threads
Array Lock
Clh Lock
Pthread Lock
Tas Lock
Ttas Lock
No Sync


Throughput (Kops / second)
400
350
300
250
200
150
12
4
8
16
32
64
Number of threads
Array Lock
Clh Lock
Pthread Lock
Tas Lock
Ttas Lock


Throughput (Kops / second)
17500
15000
12500
10000
7500
5000
2500
0
12
4
8
16
32
64
Number of threads
Array Lock
Clh Lock
Pthread Lock
Tas Lock
Ttas Lock
No Sync


52.5
50.0
47.5
45.0
42.5
40.0
37.5
35.0
32.5
Throughput (Kops / second)
1 2 4 8 16 32 64
Number of threads
Array Lock
Clh Lock
Pthread Lock
Tas Lock
Ttas Lock


2500
2000
1500
1000
500
0
Throughput (Kops / second)
1 2 4 8 16 32 64
Number of threads
Array Lock
Clh Lock
Pthread Lock
Tas Lock
Ttas Lock
No Sync

Συμπεράσματα

Αρχικά, όπως σημειώθηκε και προηγουμένως, οι μετρήσεις που πραγματοποιήθηκαν δίχως κλείδωμα είναι εξωκοσμικές (με μια μικρή εξαίρεση στην 16άρα λίστα, όπου για έως και 4 threads τα υπόλοιπα κλειδώματα ακολουθούν το ίδιο scale). Βέβαια αυτό δεν έχει μεγάλη σημασία, εφόσον δίχως κλείδωμα η εφαρμογή μας θα υπέπιπτε σε λάθη, για παράδειγμα σε περίπτωση εγγραφής στη λίστα. Άρα οι μετρήσεις αυτές έχουν αξία περισσότερο για να κατανοήσουμε την πολύ μεγάλη επιβάρυνση που προκαλεί η χρήση κλειδωμάτων στην εφαρμογή μας.
Ας εξετάσουμε τώρα την απόδοση των 5 κλειδωμάτων. Προφανώς κανένα εξ' αυτών δεν κάνει scale(πέραν της εξαίρεσης που αναφέρθηκε, η οποία όμως οφείλεται αποκλειστικά στο μικρό μέγεθος της λίστας). Αυτό είναι αναμενόμενο διότι με την αύξηση των threads, αυξάνονται και οι διεκδικητές του εκάστοτε lock, αναιρώντας την επιπρόσθετη δυναμική που θα μας προσέφεραν.
Τα **tas_lock**, **pthread_lock** έχουν με αρκετή διαφορά τη χειρότερη απόδοση, με το δεύτερο να φαίνεται ελαφρώς καλύτερο για πολύ μεγάλες λίστες(8192). Το **tas_lock** χάνει από το γεγονός ότι με τις συνεχείς κλήσεις της συναρτησης testAndSet() δημιουργεί συμφόρηση στο bus της κρυφής μνήμης, καθυστερώντας όλα τα threads, ακόμα και αυτά που δεν ζητάνε το lock. Ακόμη, κάθε κλήση της testAndSet() από κάποιο thread, ακόμα και αποτυχημένη, κάνει invalidate τα αντίγραφα του κλειδώματος για όλα τα υπόλοιπα threads, αναγκάζοντάς τα να ξαναζητήσουν από τη μνήμη ένα νέο(συνήθως πανομοιότυπο με το προηγούμενο) αντίγραφο του κλειδώματος. Το **pthread_lock** προσφέρεται για critical sections μικρού μεγέθους, καθώς σε αντίθετη περίπτωση όλα τα υπόλοιπα threads που περιμένουν να πάρουν το lock έχουν πολύ αυξημένο νεκρό χρόνο.
Το **ttas_lock** έρχεται τρίτο, υπερέχοντας μακράν του σπάταλου προκατόχου του και μάλιστα σχεδόν κάνοντας match την απόδοση του κυρίαρχου κλειδώματος(**clh_lock)** για μεγάλες λίστες(8192) και 64 threads. Αυτό αποδίδεται προφανώς στο γεγονός ότι πλέον η testAndSet() καλείται μόνο όταν σιγουρευτούμε ότι το lock είναι ξεκλείδωτο. Και πάλι βέβαια προκαλείται συνωστισμός στο διάδρομο, ο

οποίος δύναται να μειωθεί περαιτέρω με προσθήκη exponential backoff. Το **array_lock** έρχεται κατά μέσο όρο δεύτερο, χάνοντας ωστόσο στα 64 threads από το **ttas_lock**. Στα θετικά του, το γεγονός ότι τα threads εξαρτώνται μόνο από αυτά που εκτελέστηκαν ακριβώς πριν. Στα αρνητικά του, ότι το κλείδωμα καταλαμβάνει αυξημένο χώρο στις caches των threads, ενώ είναι ιδιαίτερα επιρρεπές στο false sharing(ξανασυναντήσαμε το φαινόμενο στο πρώτο ζήτημα). Τέλος, το **clh_lock** διακρίνεται ιδιαίτερα στις μεγαλύτερες λίστες, καθώς κατορθώνει να απομονώσει τα διάφορα threads (γνωρίζουν μόνο τον predecessor τους) και να μη δημιουργεί τόσο μεγάλο συνωστισμό στο bus.

Ζήτημα 3 - Τακτικές Συγχρονισμού για Δομές Δεδομένων

Εισαγωγή

Σε αυτό το ζήτημα μας δίνεται και πάλι παραλληλοποιημένος κώδικας για την ταυτόχρονη υλοποίηση διαφόρων operations σε μια συνδεδεμένη λίστα από διάφορα threads. Εμείς καλούμαστε να:

1) Υλοποιήσουμε λίστες με συγκεκριμένες τακτικές συγχρονισμού.
2) Εξετάσουμε την απόδοση αυτών για διάφορους συνδυασμούς threads - μεγεθών λίστας, καθώς και 2 διαφορετικά σενάρια operations.

Υλοποίηση

Παρακάτω θα επιδείξουμε τις λίστες που υλοποιήσαμε(σε όλες τις περιπτώσεις δείχνουμε μόνο τα κομμάτια που δημιουργήσαμε/τροποποιήσαμε). Σε όλες τις λίστες, οι ζητούμενες μεθόδοι *add()*, *contains()* και *remove()* έχουν υλοποιηθεί με βάση τις

διαφάνειες του μαθήματος, οπότε δεν θα αναλυθούν. Ξεκινάμε με την **ll_fgl**, η οποία χρησιμοποιεί fine grained synchronization.

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

struct linked_list {
    ll_node_t *head;
    pthread_spinlock_t lock;
};

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;
}

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;
}

int ll_contains(ll_t *ll, int key)
{
    ll_node_t *curr, *prev;
    int ret = 0;
    while(pthread_spin_trylock(&ll->lock) != 0);

    prev = ll->head;
    while(pthread_spin_trylock(&prev->lock) != 0);
    pthread_spin_unlock(&ll->lock);

    curr = prev->next;
```

```
    while(pthread_spin_trylock(&curr->lock) != 0);

    while(curr->key < key) {
        pthread_spin_unlock(&prev->lock);
        prev = curr;
        curr = curr->next;
        if(curr)
            while(pthread_spin_trylock(&curr->lock) != 0);
    }

    pthread_spin_unlock(&curr->lock);
    pthread_spin_unlock(&prev->lock);

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

int ll_add(ll_t *ll, int key)
{
    ll_node_t *curr, *prev;
    ll_node_t *new_node;
    int ret = 0;
    while(pthread_spin_trylock(&ll->lock) != 0);

    prev = ll->head;
    while(pthread_spin_trylock(&prev->lock) != 0);
    pthread_spin_unlock(&ll->lock);

    curr = prev->next;
    while(pthread_spin_trylock(&curr->lock) != 0);

    while(curr->key < key) {
        pthread_spin_unlock(&prev->lock);
        prev = curr;
        curr = curr->next;
        if(curr)
            while(pthread_spin_trylock(&curr->lock) != 0);
    }

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

        new_node->next = curr;
        prev->next = new_node;
    }

    pthread_spin_unlock(&curr->lock);
    pthread_spin_unlock(&prev->lock);
```

```
    return ret;
}

int ll_remove(ll_t *ll, int key)
{
    int ret = 0;
    ll_node_t *curr, *prev;
    while(pthread_spin_trylock(&ll->lock) != 0);

    prev = ll->head;
    while(pthread_spin_trylock(&prev->lock) != 0);
    pthread_spin_unlock(&ll->lock);

    curr = prev->next;
    while(pthread_spin_trylock(&curr->lock) != 0);

    while(curr->key < key) {
        pthread_spin_unlock(&prev->lock);
        prev = curr;
        curr = curr->next;
        if(curr)
            while(pthread_spin_trylock(&curr->lock) != 0);
    }

    if(key == curr->key) {
        ret = 1;
        prev->next = curr->next;
        ll_node_free(curr);
    }

    if(ret == 0) {
        pthread_spin_unlock(&curr->lock);
    }
    pthread_spin_unlock(&prev->lock);

    return ret;
}
```

Στα structs *ll_node*, *linked_list* έχουμε προσθέσει από ένα κλείδωμα, απαραίτητο για την υλοποίηση του hand-over-hand locking στις μεθόδους *add()*, *contains()* και *remove()*.

Συνεχίζουμε με την **ll_opt**, η οποία χρησιμοποιεί optimistic synchronization.

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

struct linked_list {
    ll_node_t *head;
};

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;
}

int validate(ll_t *ll, ll_node_t *prev, ll_node_t *curr) {
    ll_node_t *node = ll->head;
    while(node->key <= prev->key) {
        if(node == prev) {
            return (prev->next == curr);
        }
        node = node->next;
    }
    return 0;
}

int ll_contains(ll_t *ll, int key) {
    ll_node_t *curr, *prev;
    int ret = 0;
    bool overAndOut = false;

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

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

        while(pthread_spin_trylock(&prev->lock) != 0);
        while(pthread_spin_trylock(&curr->lock) != 0);

        if(validate(ll, prev, curr) == 1) {
```

```
            ret = (curr->key == key);
            overAndOut = true;
        }

        pthread_spin_unlock(&prev->lock);
        pthread_spin_unlock(&curr->lock);

        if(overAndOut)
            break;
    }
    return ret;
}

int ll_add(ll_t *ll, int key)
{
    ll_node_t *curr, *prev;
    ll_node_t *new_node;
    int ret = 0;
    bool overAndOut = false;

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

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

        while(pthread_spin_trylock(&prev->lock) != 0);
        while(pthread_spin_trylock(&curr->lock) != 0);

        if(validate(ll, prev, curr) == 1) {
            if(curr->key != key) {
                ret = 1;
                new_node = ll_node_new(key);
                prev->next = new_node;
                new_node->next = curr;
            }
            overAndOut = true;
        }

        pthread_spin_unlock(&prev->lock);
        pthread_spin_unlock(&curr->lock);

        if(overAndOut)
            break;
    }
    return ret;
}
```

```c
int ll_remove(ll_t *ll, int key)
{
    ll_node_t *curr, *prev;
    int ret = 0;
    bool overAndOut = false;

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

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

        while(pthread_spin_trylock(&prev->lock) != 0);
        while(pthread_spin_trylock(&curr->lock) != 0);

        if(validate(ll, prev, curr) == 1) {
            if(curr->key == key) {
                ret = 1;
                prev->next = curr->next;
            }
            overAndOut = true;
        }

        pthread_spin_unlock(&prev->lock);
        pthread_spin_unlock(&curr->lock);

        if(overAndOut)
            break;
    }

    return ret;
}
```

Τέλος, δίνουμε την **ll_lazy**, που χρησιμοποιεί lazy synchronization.

```c
typedef struct ll_node {
    int key;
    struct ll_node *next;
    pthread_spinlock_t lock;
    bool deleted;
} ll_node_t;

static ll_node_t *ll_node_new(int key)
```

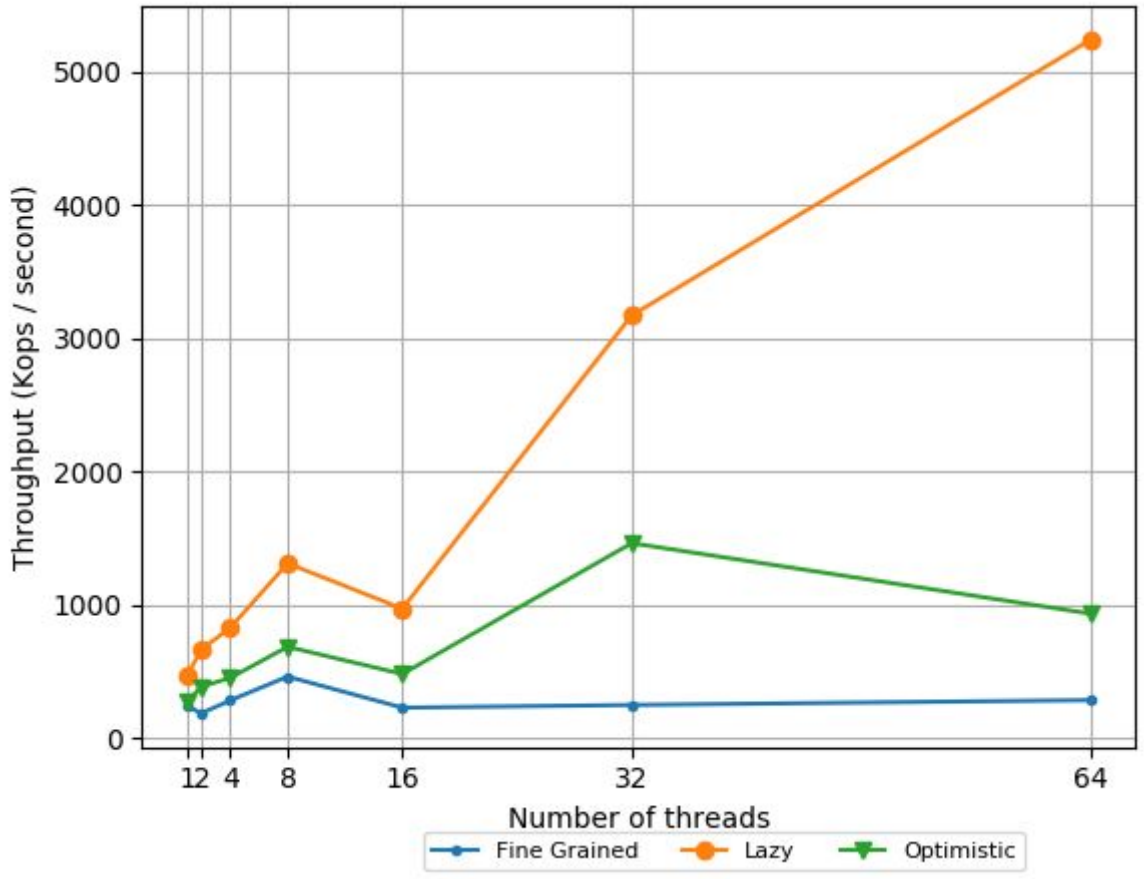
```
{
    ll_node_t *ret;

    XMALLOC(ret, 1);
    ret->key = key;
    ret->next = NULL;
    ret->deleted = false;
    pthread_spin_init(&ret->lock, PTHREAD_PROCESS_SHARED);

    return ret;
}

int validate(ll_node_t *prev, ll_node_t *curr) {
    return (!prev->deleted && !curr->deleted && prev->next == curr);
}

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

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

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

int ll_add(ll_t *ll, int key)
{
    ll_node_t *curr, *prev;
    ll_node_t *new_node;
    int ret = 0;
    bool overAndOut = false;

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

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

        while(pthread_spin_trylock(&prev->lock) != 0);
        while(pthread_spin_trylock(&curr->lock) != 0);

        if(validate(prev, curr) == 1) {
            if(curr->key != key) {
                ret = 1;
```

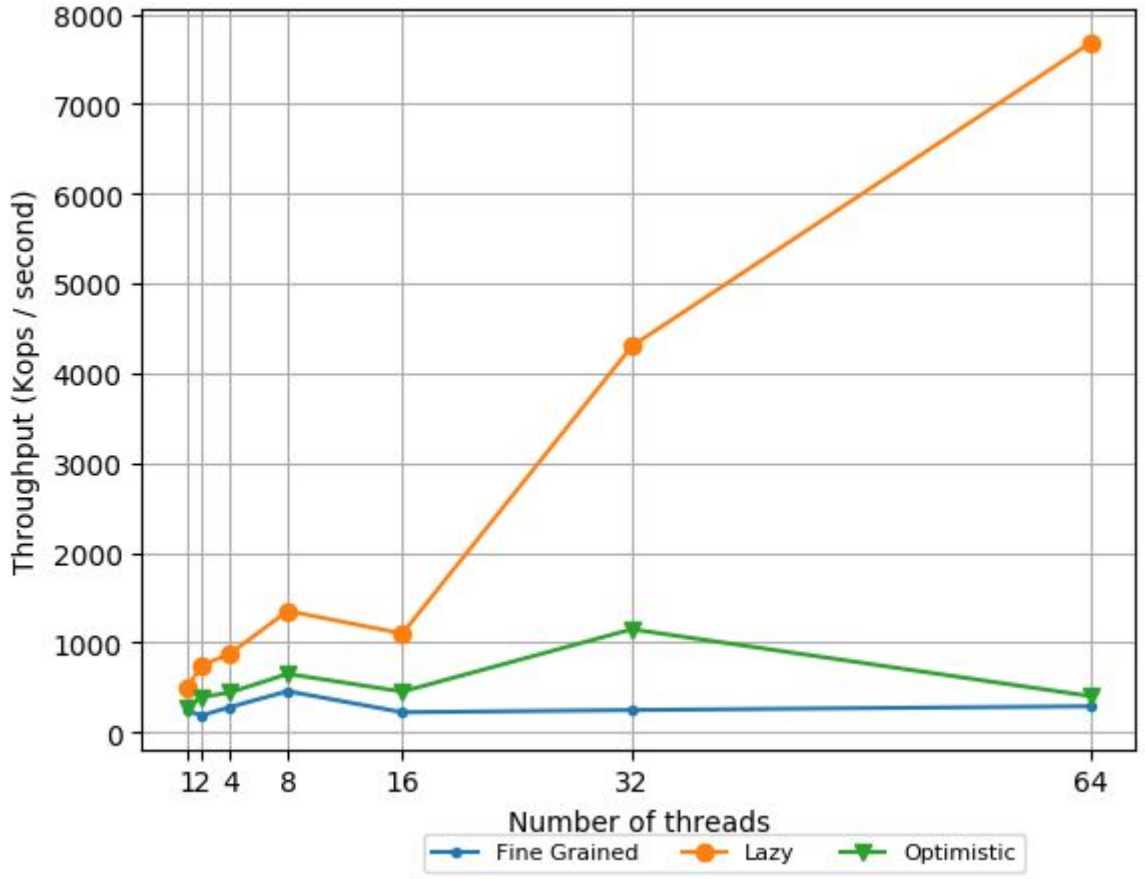
```
                new_node = ll_node_new(key);
                new_node->next = curr;
                prev->next = new_node;
            }
            overAndOut = true;
        }

        pthread_spin_unlock(&prev->lock);
        pthread_spin_unlock(&curr->lock);

        if(overAndOut)
            break;
    }

    return ret;
}

int ll_remove(ll_t *ll, int key)
{
    ll_node_t *curr, *prev;
    int ret = 0;
    bool overAndOut = false;

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

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

        while(pthread_spin_trylock(&prev->lock) != 0);
        while(pthread_spin_trylock(&curr->lock) != 0);

        if(validate(prev, curr) == 1) {
            if(curr->key == key) {
                ret = 1;
                curr->deleted = 1;
                prev->next = curr->next;
            }
            overAndOut = true;
        }

        pthread_spin_unlock(&prev->lock);
        pthread_spin_unlock(&curr->lock);

        if(overAndOut)
            break;
    }
```

```
    return ret;
}
```

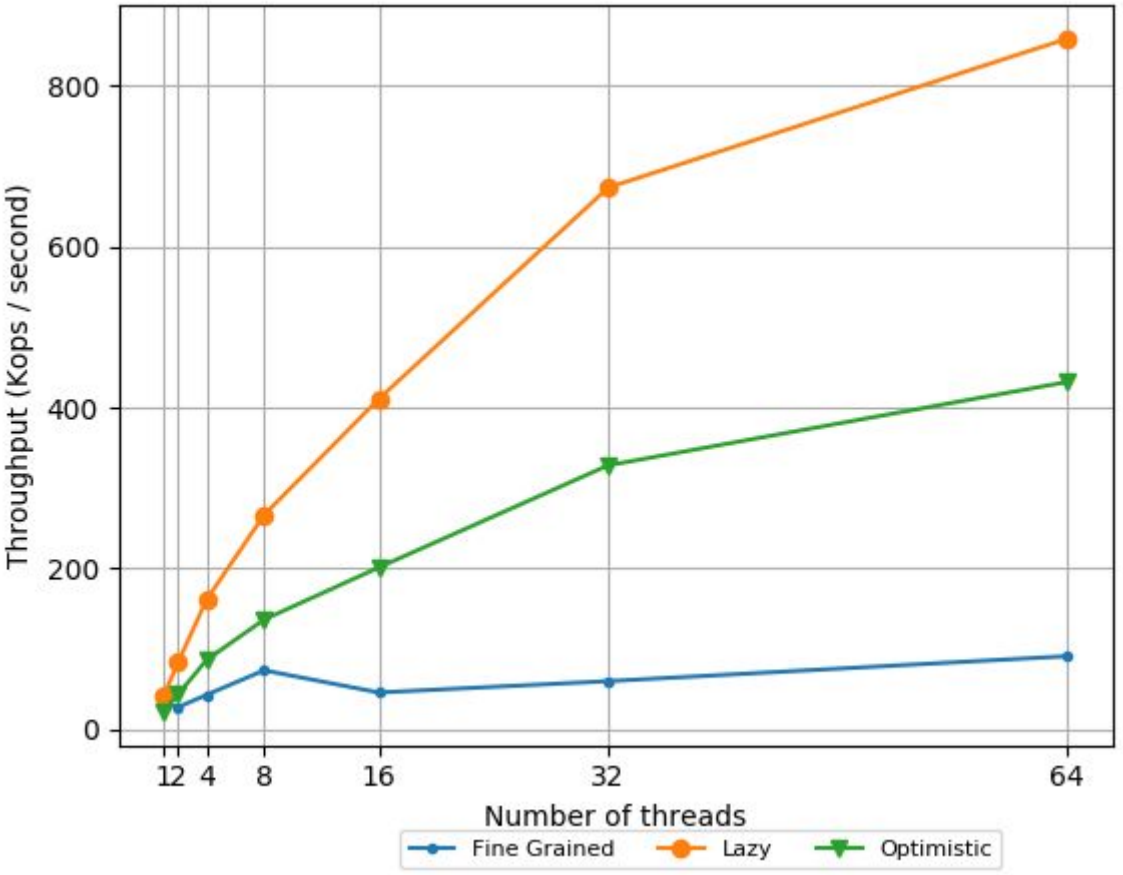
Εκτέλεση - Επίδειξη Μετρήσεων

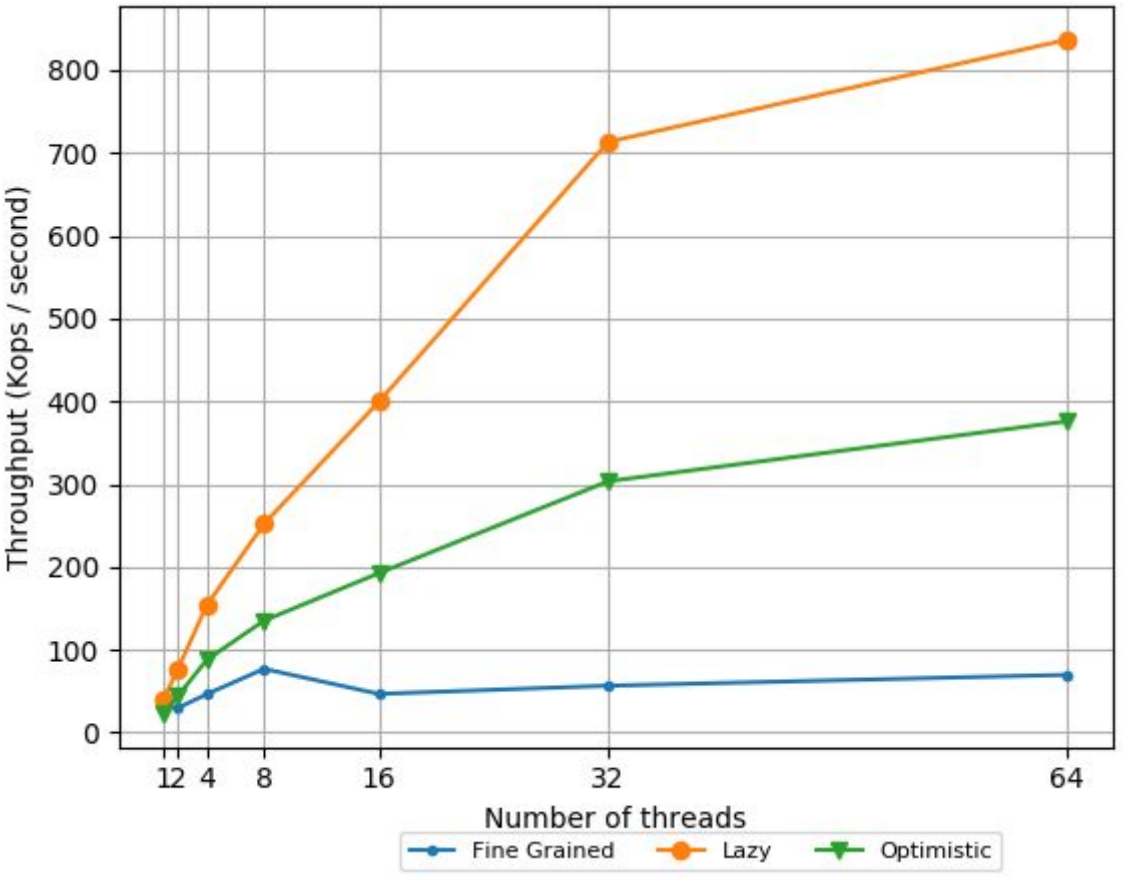
Τόσο για την εκτέλεση των μετρήσεων όσο και την παραγωγή των γραφημάτων, χρησιμοποιήθηκαν αντίστοιχα script με αυτά που έχουν ήδη παρουσιαστεί στο 2ο ζητούμενο της άσκησης.
Παρακάτω, δίνουμε τα throughput που προέκυψαν με τη χρήση των διαφόρων υλοποιήσεων της λίστας. Αρχικά, ακολουθούν τα γραφήματα για λίστα 1024 στοιχείων και αναλογίες operations 20-40-40 και 80-10-10 αντίστοιχα.





Στη συνέχεια, δίνονται οι γραφικές για λίστα 8192 στοιχείων και αναλογίες operations 20-40-40 και 80-10-10 αντίστοιχα.


Throughput (Kops / second)
800
600
400
200
0
1
2
4
8
16
32
64
Number of threads
Fine Grained
Lazy
Optimistic


Throughput (Kops / second)
800
700
600
500
400
300
200
100
0
1
2
4
8
16
32
64
Number of threads
Fine Grained
Lazy
Optimistic

Συμπεράσματα

Η ιεράρχηση των 3 υλοποιήσεων είναι ξεκάθαρη, ιδιαίτερα στη λίστα των 8192 στοιχείων. Η **ll_fgl** αποδίδει το ίδιο (άσχημα), ανεξάρτητα των αναλογιών των operations, πράγμα λογικό εφόσον όλες εφαρμόζουν hand-over-locking. Η **ll_opt** από την άλλη, υποφέρει στα σενάρια όπου το 80% των operations είναι *contains()*. Το συγκεκριμένο στατιστικό είναι αποθαρρυντικό, αφού κατά μέσο όρο το 90% των κλήσεων σε μια συνδεδεμένη λίστα είναι *contains()*. Επίσης, προφανής προϋπόθεση είναι το κόστος της διπλής διάσχισης της λίστας(που απαιτείται από την *validate()*) να είναι μικρότερο από αυτό της απλής διάσχισης της λίστας με hand-over-locking. Τέλος, η **ll_lazy** δεν χρησιμοποιεί κλειδώματα στην *contains()*, γεγονός που εκτοξεύει την απόδοσή της στα προαναφερθέντα σενάρια.