μαρια ηλεκτρα γαργαλα – Α.μ.4330

Δημοσθενησ ζαγκασ – α.μ.4359

Email: cs04330@uoi.gr

Email: cs04359@uoi.gr

ετοσ: 2020-2021

1η Εργαστηριακη ασκηση  
-Λειτουργικα συστηματα

**Γενική περιγραφή προγράμματος:**

Το project που μας ανατέθηκε, υλοποιεί μία μηχανή αποθήκευσης δεδομένων (key-value store). Βασίζεται στην δομή LSM-Tree, μία δομή που διατηρεί ταξινομημένα τα δεδομένα εσωτερικά στην μνήμη και στον δίσκο, και αποθηκεύει και αναζητά ζευγάρια κλειδιού-τιμής. Τα ταξινομημένα δεδομένα βοηθούν στην πολυπλοκότητα, γιατί μπορούμε να τα εγγράψουμε και να τα αναζητήσουμε πιο γρήγορα.

Το LSM-Tree αποτελείται από δύο δομές, την Memtable και την SST. Η δομή Memtable υλοποιείται με μία δομή δεδομένων με πολλές λίστες και βρίσκεται πάντα στην μνήμη. Ουσιαστικά, μία κλήση για εισαγωγή κλειδιού-τιμής (db\_add()) εισάγεται στο Memtable. Αν κάνω κλήση της db\_get() αναζητά αρχικά την εγγραφή στο Memtable και εφόσον δε βρεθεί θα την αναζητήσει στο SST. Σε περίπτωση που το Memtable γεμίσει, θα γίνει merge (στον δίσκο) για να μεταφερθούν τα αρχεία στο δίσκο.

Στην Memtable υπάρχει κι ένα αρχείο log όπου αποθηκεύει στον δίσκο τις εγγραφές. Η ύπαρξη του αρχείου log, χρησιμεύει σε περιπτώσεις σφαλμάτων, παραδείγματος χάριν όταν κρασάρει ο υπολογιστής μας ή κλείσει καταλάθος.

Η εντολή put εισάγει στο Memtable και στο log αρχείο ένα ζευγάρι κλειδί-τιμή στην βάση. Σε περίπτωση που το log αρχείο γεμίσει, το Memtable παγώνει, το γράφουμε στο δίσκο και φτιάχνουμε ένα καινούριο Memtable για τις επόμενες εγγραφές.

Η δομή SST βρίσκεται στον δίσκο και έχει πολλά επίπεδα (έξι), όπου το καθένα από αυτά περιέχει αρχεία με ταξινομημένες εγγραφές. Σε περίπτωση που τα αρχεία οποιουδήποτε επιπέδου γίνουν πάρα πολύ μεγάλα ή όταν το πλήθος των αρχείων είναι πολύ μεγάλο, γίνεται συγχώνευση (combaction) και το αρχείο αυτό πηγαίνει στο επόμενο επίπεδο.

Η εντολή get αναζητά πρώτα στο Memtable τις εγγραφές και αν δεν βρεθούν, τότε τις αναζητά στο SST (σε όλα τα επίπεδα: level0-level6).

# **Υλοποίηση**

**Γενική περιγραφή:**

Στόχος είναι η παραλληλοποίηση των functions \_write\_test και \_read\_test το οποίο επιτυγχάνεται με την δημιουργία νημάτων στην bench.c, καθώς και ο παραλληλισμός της db\_get().

**Αναλυτική Εξήγηση:**

Αρχικά ζητούμε στην γραμμή εντολών τον αριθμό των αιτήσεων για read και για write και τον αριθμό των νημάτων που επιθυμεί ο χρήστης. Ο χρήστης μπορεί να επιλέξει και τις λειτουργίες write Ή read, γράφοντας και τον αριθμό των αιτημάτων για την λειτουργία που επέλεξε (./a.out write 1000 Ή ./a.out read 300).

Παράδειγμα: ./a.out read\_write 600 50 35 : επιλογή read\_write, 600 αιτήσεις για read, 50 αιτήσεις για write και 35 νήματα. Με βάση τον αριθμό των αιτήσεων για τις λειτουργίες read και write χωρίζουμε και τα νήματα, ώστε να βελτιώσουμε την απόδοση. Όταν δηλαδή έχω 9000 write και 1000 read, θα χωρίζουμε τα νήματα ώστε να εκμεταλλευτεί τα περισσότερα η write, που έχει τον μεγαλύτερο αριθμό αιτήσεων. Αυτό έχει σκοπό την βελτιστοποίηση του χρόνου, αφού είναι λογικό η συνάρτηση με τις περισσότερες αιτήσεις να έχουν ανάγκη περισσότερα νήματα.

*Bench.h:*

* Γραμμή 8-12: Προσθήκη #include <pthread.h> & #include "../engine/db.h" & #define DATAS ("testdb"). Ο λόγος εξηγείται παρακάτω.
* Γραμμή 25-27: Αρχικοποίηση των πρωτότυπων συναρτήσεων για **\_write\_test**, **\_read\_test** και **statistics** (θα εξηγηθεί η χρήση της παρακάτω)

**void statistics(void \* temp,int flag);**

**void \*\_write\_test(void \* args\_write);**

**void \*\_read\_test(void \* args\_read);**

* Γραμμή 30-36: Ορισμός του struct. Οι μεταβλητές της δομής είναι **long int count** = **long int write\_count;** + **long int read\_count;** και **int r**; και **DB\* db**;

Οι παράμετροι των **\_write\_test** και **\_read\_test** τροποποιήθηκαν κατάλληλα λόγω της χρήσης της pthread\_create(..). Πλέον οι παράμετροί τους εισέρχονται μέσω της struct που δημιουργήσαμε.

**struct args{**

**long int count;  
long int write\_count;  
long int read\_count;  
int r;  
DB\* db;**

**};**

*Bench.c:*

* Γραμμή 21: Προσθήκη της printf(numOfThreads) στο print\_header για να τυπώνεται ο αριθμός των δοθέντων νημάτων στην περίπτωση που ο χρήστης έχει επιλέξει \_read\_write. Επίσης, η \_print\_header παίρνει σαν παράμετρο από την κλήση της και το numOfThreads.
* Γραμμή 73-83: Ορισμός των απαραίτητων μεταβλητών.

**DB\* db; int flag;**

**long int numOfThreads\_write;**

**long int numOfThreads\_read;**

**long int numOfThreads;**

**long int read\_count;**

**long int write\_count;**

**long int ceiling;**

**long double num;**

**struct args thread\_args;**

**thread\_args.r = 0;**

* Γραμμή 73: Ορισμός της db. Η βάση θέλουμε να ανοίγει/κλείνει μία φορά κι όχι από κάθε thread, γι’ αυτό και την απομονώσαμε από την kiwi.c και καλείται πλέον στην bench.c. Η db ανοίγει μία φορά(γραμμές 154,174,188) και στα τρία if ξεχωριστά (για read,write και read\_write) και τυπώνει τις λεπτομέρειές της, ενώ κλείνει μία φορά στο τέλος του προγράμματος (χρειάστηκε να κάνουμε #include "../engine/db.h" και #define DATAS ("testdb") στην bench.h).
* Γραμμή 85-89: Έλεγχος με if για το εάν έχει δεχθεί τα κατάλληλα ορίσματα από το terminal.
* Γραμμή 92-94: Έλεγχος των ορισμάτων της main με if για το αν θα εκτελέσουμε read, write ή read\_write.
* Γραμμή 95-107: Το argc > 3, άρα ζητάμε να κάνει read\_write, οπότε και αρχικοποιούμε όλες τις μεταβλητές από τα ορίσματα της main. Το argv[2] αναφέρεται στις αιτήσεις της read, το argv[3] αναφέρεται στις αιτήσεις της write και το argv[4] στον αριθμό των νημάτων. Ορίζουμε και τις παραμέτρους του struct αντίστοιχα.

**write\_count = atoi(argv[3]);**

**thread\_args.write\_count = write\_count;**

**read\_count = atoi(argv[2]);**

**thread\_args.read\_count = read\_count;**

**thread\_args.count = write\_count + read\_count;**

**numOfThreads = atoi(argv[4]);**

* Γραμμή 106-141: Χωρισμός των threads. Θέλουμε να χωρίζονται τα νήματα ανάλογα με τον αριθμό των αιτήσεων. Για παράδειγμα για 60 write και 40 read με 10 νήματα, 6 νήματα θα χρησιμοποιηθούν στην write και 4 στην read.

*Μία κρίσιμη περίπτωση όπου δεν γινόταν σωστά ο χωρισμός των νημάτων:* Έχουμε read = 4, write = 8, threads = 4. Ο επιθυμητός χωρισμός των νημάτων θα είναι 3 για write και 1 για read. Παρ’ όλα αυτά το ποσοστό (write/(read+write))\*threads είναι 2.9, και λόγω του integer(2.9) είναι 2 και επομένως χάνουμε ένα thread. Για τον αποφυγή του προβλήματος αυτού, αρχικοποιούμε μια μεταβλητή **double num;** όπου δίνεται το ποσοστό των νημάτων για write και παίρνουμε τεχνητά το **int ceiling;** της **num**(χωρίς χρήση της βιβλιοθήκης math.h), το οποίο θα ισούται με 3. Κάνοντας μία αφαίρεση των νημάτων για write από τα ολικά νήματα, παίρνουμε τον αριθμό των νημάτων για read.

*Μία άλλη κρίσιμη περίπτωση είναι:*

Ο υπολογισμός του ποσοστού να έχει αποτέλεσμα μικρότερο του 1 (πχ 0.3) όπου για integer(0.3) θα γίνει 0, το οποίο δεν είναι αποδεκτό. Σ’ αυτή την περίπτωση ορίζουμε 1 thread (πχ για το read) και αφαιρούμε 1 από τα threads για write.

**if((ceiling - num)<0.5){**

**numOfThreads\_write = ceiling;**

**}**

**else{**

**numOfThreads\_write = (int)num;**

**if(numOfThreads\_write == 0){**

**numOfThreads\_write = 1;**

**}**

**}**

**numOfThreads\_read = numOfThreads-numOfThreads\_write;**

**if(numOfThreads\_read == 0){**

**numOfThreads\_read = 1;**

**numOfThreads\_write--;**

**}**

**}**

* Γραμμή 148-161: Βρισκόμαστε στην περίπτωση όπου πραγματοποιείται μόνο η write, **threag\_args.write\_count = thread\_args.count** αφού όλα τα count είναι για το write. Ορίζουμε flag=0 που θα χρησιμοποιηθεί αργότερα στην statistics όπου και θα αναλυθεί.

Επίσης στην συνάρτηση \_printf\_header περνάμε σαν δεύτερο όρισμα το 0, αφού δεν έχουμε νήματα.

Την μεταβλητή r, αφού πλέον την περνάμε από τον struct, την χρησιμοποιούμε στην Γραμμή 156-157 όπως φαίνεται:

**if (argc == 4){**

**thread\_args.r = 1;**

**}**

* Γραμμή 164-176: Βρισκόμαστε στην περίπτωση για read και πραγματοποιούνται οι αντίστοιχες ενέργειες όπως στην write.  
  Γραμμή 172: Ορίζουμε την thread\_args.write\_count = 0 ώστε να προσπεράσουμε το pthread\_cond\_wait όταν εκτελούμε μόνο read.
* Γραμμή 179-236: Βρισκόμαστε στην περίπτωση read\_write. Στην Γραμμή 184-185 καλούμε τις συναρτήσεις \_print\_header & \_print\_environment ώστε να εκτυπώνονται οι πληροφορίες για την κεφαλίδα και το σύστημα. Στην Γραμμή 188-190 έχουμε βάλει αντίστοιχη περίπτωση για **thread\_args.r = 6;**, όπως κάνουμε και στις άλλες δύο περιπτώσεις. Στην Γραμμή 191 κάνουμε δυναμική δέσμευση μνήμης για τα νήματα.

**newthread = (pthread\_t \*)malloc(numOfThreads \* sizeof(pthread\_t));**

**if(newthread == NULL){ //check if malloc worked**

**printf("Malloc failed!\n");**

**exit(1);**

**}**

Στην Γραμμή 204-231 καλούμε τις συναρτήσεις \_write\_test και \_read\_test μέσω των νημάτων.

Η δημιουργία των νημάτων θέλουμε να γίνει παράλληλα ως έναν βαθμό κι όχι σειριακά (όχι δηλαδή πρώτα όλα τα νήματα για write και μετά όλα τα νήματα για read, σε δύο for). Αντίθετα, θα θέλαμε σε μια for, σε κάθε επανάληψη της, να δημιουργούμε νήματα για write και για read. Αυτό πραγματοποιείται με τον παρακάτω κώδικα, που θα τον εξηγήσουμε με ένα παράδειγμα.

Έστω ότι έχουμε 4 νήματα για write και 6 για read. Δημιουργούμε με μία for παράλληλα τα 4 νήματα για write και read και στην συνέχεια με μία δεύτερη for δημιουργούμε τα δύο υπολειπόμενα νήματα για read.

**if(numOfThreads\_read > numOfThreads\_write){**

**for(i=0; i<numOfThreads\_write;i++){**

**pthread\_create(&newthread[i],NULL,\_write\_test,(void \*) &thread\_args);**

**pthread\_create(&newthread[i+numOfThreads\_write],NULL, \_read\_test,(void \*) &thread\_args);**

**}**

**for(i=numOfThreads\_write; i<(numOfThreads - numOfThreads\_write);i++){**

**pthread\_create(&newthread[i+numOfThreads\_write],NULL,\_read\_test,(void \*) &thread\_args);**

**}**

**}**

Το αντίστοιχο συμβαίνει όταν έχουμε περισσότερα νήματα για write αντί για read, ενώ αν είναι ίσα χρησιμοποιούμε μόνο μία for.

Στην Γραμμή 234-237 κάνουμε χρήση της join για να περιμένουμε όλα τα νήματα πριν τερματίσουμε.

* Γραμμή 245: Κλείνουμε την βάση.
* Γραμμή 246: Καλούμε την statistics με ορίσματα το struct thread\_args (περιέχει τον αριθμό των write/read, το count και την μεταβλητή db) και το flag.

*Kiwi.c:*

* Γραμμή 8-9: Ορίζουμε τα mutex\_lock.
* Γραμμή 11: Ορίζουμε την μεταβλητή συνθήκης (condition Variable)
* Γραμμή 13-29: Ορίζουμε/αρχικοποιούμε κάποιες μεταβλητές που θα χρειαστούν στην συνέχεια
* Γραμμή 31-106 write\_test:  
  Στην Γραμμή 31, για την δημιουργία νημάτων, φέρνουμε τη συνάρτηση \_write\_test στην μορφή void\* \_write\_test(void \* args). Στην Γραμμή 41-44 δημιουργούμε ένα τοπικό struct args, μέσω του οποίου αντιγράφουμε τις μεταβλητές (count,r,db). Αυτό το κάναμε για να μην αλλάξουμε σε όλον τον κώδικα τις μεταβλητές αυτές με αυτές που ορίζουμε στην bench.

Στην Γραμμή 59-66, για να υπολογίσουμε το συνολικό κόστος χρειαζόμαστε το start από το πρώτο νήμα, που σηματοδοτεί την έναρξη. Χωρίς κάποια τροποποίηση του κώδικα, κάθε νήμα που εκτελούσε την συνάρτηση άλλαζε την τιμή του start. Γι’ αυτό με την χρήση αμοιβαίου αποκλεισμού παίρνουμε το πρώτο start μέσω της if και τα υπόλοιπα νήματα δεν μπορούν να το αλλάξουν.

**pthread\_mutex\_lock(&mymutexW);**

**if(k == 0){**

**startW = get\_ustime\_sec();  
 k = 1;  
}  
pthread\_mutex\_unlock(&mymutexW);**

Την μεταβλητή start την ορίσαμε ως global για να είναι καθολική η τιμή της.

Στην Γραμμή 70 δημιουργούμε το mutex\_lock όπου περιμένουν όλα τα νήματα πριν την εκτέλεση του db\_add.  
Στην Γραμμή 71, κάνουμε αύξηση του global j(counter) για να παίρνει το επόμενο νήμα την ενημερωμένη τιμή.  
Στην Γραμμή 72 ελέγχουμε αν το j είναι μεγαλύτερο από το count κι αν ισχύει κάνουμε break (αφού θα έχουμε εκτελέσει όλες τις add), αλλιώς συνεχίζουμε.

Στην Γραμμή 90 γίνεται το db\_add.  
Στην Γραμμή 91 ελέγχουμε αν το j ισούται με το count∙ τότε στέλνουμε σήμα ώστε να αρχίσουν την λειτουργία τους όλα τα νήματα (**pthread\_cond\_broadcast(&condVar);**) στην read (θα εξηγηθεί στην read), και αυξάνουμε το done κατά ένα.

**db\_add(db, &sk, &sv);**

**if(j == count){**

**pthread\_cond\_broadcast(&condVar);**

**done = 1;**

**}**

Στην Γραμμή 105, παίρνουμε από το τελευταίο νήμα τον χρόνο λήξης της write, αποθηκεύοντάς το στην καθολική μεταβλητή **long long int endW;**.

Στην Γραμμή 106 παίρνουμε το συνολικό κόστος: **costW = endW -startW;**.

* Γραμμή 109-185 read\_test:  
  Στην Γραμμή 111, για την δημιουργία νημάτων, φέρνουμε τη συνάρτηση \_read\_test στην μορφή void\* \_read\_test(void \* args).

Στην Γραμμή 124-127 κάνουμε την ίδια διαδικασία για την struct με αυτήν της \_write\_test.

Στην Γραμμή 129-137, δέχεται το σήμα από την \_write\_test και η τιμής της done παύει να είναι διάφορη του 1, κι έτσι απελευθερώνονται όλα τα νήματα.

**pthread\_mutex\_lock(&mymutexW);**

**while(done != 1){**

**pthread\_cond\_wait(&condVar,&mymutexW);**

**}**

**pthread\_mutex\_unlock(&mymutexW);**

Στην Γραμμή 139-145, για την απολαβή του χρόνου εκτελούμε την ίδια διαδικασία που κάναμε στην write.

Στην Γραμμή 147, η while δημιουργήθηκε παρόμοια με αυτήν της \_write\_test. Η μόνη διαφορά είναι ότι η **unlock(&mymutexR)** γίνεται στην   
  
Γραμμή 168 (πριν από την κλήση της db\_get) αφού έχουμε βάλει εσωτερικά lock στην db\_get, όπως θα περιγραφεί στην db.c.

Στην Γραμμή 173, γίνεται το unlock\_mutex του sst\_get στην db.c καθώς το lock του ίδιου mutex γίνεται μέσα στην db.c. Το unlock δε μπορεί να γίνει μέσα στο db.c καθώς το sst\_get εκτελείται στο return του db\_get.

* Γραμμή 195-230: Υλοποιείται η statistics η οποία είναι υπεύθυνη για την τύπωση των στατιστικών για κάθε μία από τις 3 περιπτώσεις(read,write, read\_write). Για flag=0 είναι η περίπτωση write και τυπώνονται τα στατιστικά μόνο για την write, για flag=1 είναι η read και τυπώνονται τα στατιστικά μόνο για την read και για flag=2 είναι η read\_write και τυπώνονται τα στατιστικά για την write και την read.

Γραμμή 198, υπολογίζεται το ολικό κόστος προσθέτοντας το κόστος της καθολικής μεταβλητής read και write.

**cost = costR+costW;**

Στην Γραμμή 196 φτιάχνουμε έναν struct για να πάρουμε τον συνολικό αριθμό των αιτήσεων (count) από τον struct που μας δίνει σαν παράμετρο η bench.c.

**struct args\* stats = (struct args\*) temp;**

* Με βάση τον ορισμό που βρήκαμε στο διαδίκτυο:  
  *Ορισμός ρυθμοαπόδοσης*: , όπου l όγκος των δεδομένων σε bits, t ο χρόνος που απαιτήθηκε για την ολοκλήρωση της μετάδοσης.

Σε κάθε read ή write, μεταφέρουμε προς/από την βάση ένα ζευγάρι κλειδί-τιμή, τα οποία είναι τύπου char, δηλαδή 1byte(=8bit) το καθένα. Επομένως το ζευγάρι είναι 16bits συνολικά. Όταν εκτελούμε read\_write, επειδή εκτελούμε και τις δύο συναρτήσεις (read και write), μεταφέρουμε 32bit σε κάθε εκτέλεση.

Άρα, η ρυθμοαπόδοση = (32\*count)/cost, όπου count είναι ο αριθμός των αιτήσεων (read και write) και cost το ολικό κόστος.

**long int totThroughput;**

**totThroughput = (32\*stats->count)/cost;**

Από την Γραμμή 214-229 κάνουμε τις κατάλληλες αλλαγές στις printf ώστε να τυπώνονται σωστά τα στατιστικά. Η ρυθμοαπόδοση θέλουμε να τυπώνεται μόνο στην περίπτωση που έχουμε read\_write (δηλαδή στην πρώτη περίπτωση που έχουμε μόνο flag==2).

* Γραμμή 206-211:

Όταν το κόστος είναι 0 απαιτώντας λίγες αιτήσεις, τότε ορίζουμε την ρυθμοαπόδοση ίση με το μηδέν, για να αποφύγουμε την διαίρεση με το μηδέν (θα τείνει άπειρο).

**if(cost == 0){**

**totThroughput = 0;**

**}**

**else{**

**totThroughput = (32\*stats->count)/cost;**

**}**

*db.h:*

* Γραμμή 10: Ορισμός του pthread\_mutex όπου θα χρησιμοποιηθεί στην kiwi.c & db.c.

**pthread\_mutex\_t mymutex\_get\_sst;**

*db.c:*

* Γραμμή 8: Αρχικοποίηση mutex

**pthread\_mutex\_t mymutex\_get\_memtable = PTHREAD\_MUTEX\_INITIALIZER;**

* Γραμμή 69-84: Η αρχική ιδέα ήταν να κλείναμε ολικά την db\_get και να την εκτελούσε ένα νήμα την φορά για αναζήτηση.

Αφού όμως οι δύο δομές είναι ανεξάρτητες, κλείνουμε το mutex πάνω από την db\_add (όπως φαίνεται στην kiwi.c), ώστε να περνάνε όλα τα νήματα στην συνάρτηση και βάζουμε mutex στην memtable\_get και την sst\_get για να δημιουργήσουμε καλύτερο παραλληλισμό.

**pthread\_mutex\_lock(&mymutex\_get\_memtable);**

**if (memtable\_get(self->memtable->list, key, value) == 1){**

**pthread\_mutex\_unlock(&mymutex\_get\_memtable);**

**return 1;**

**}**

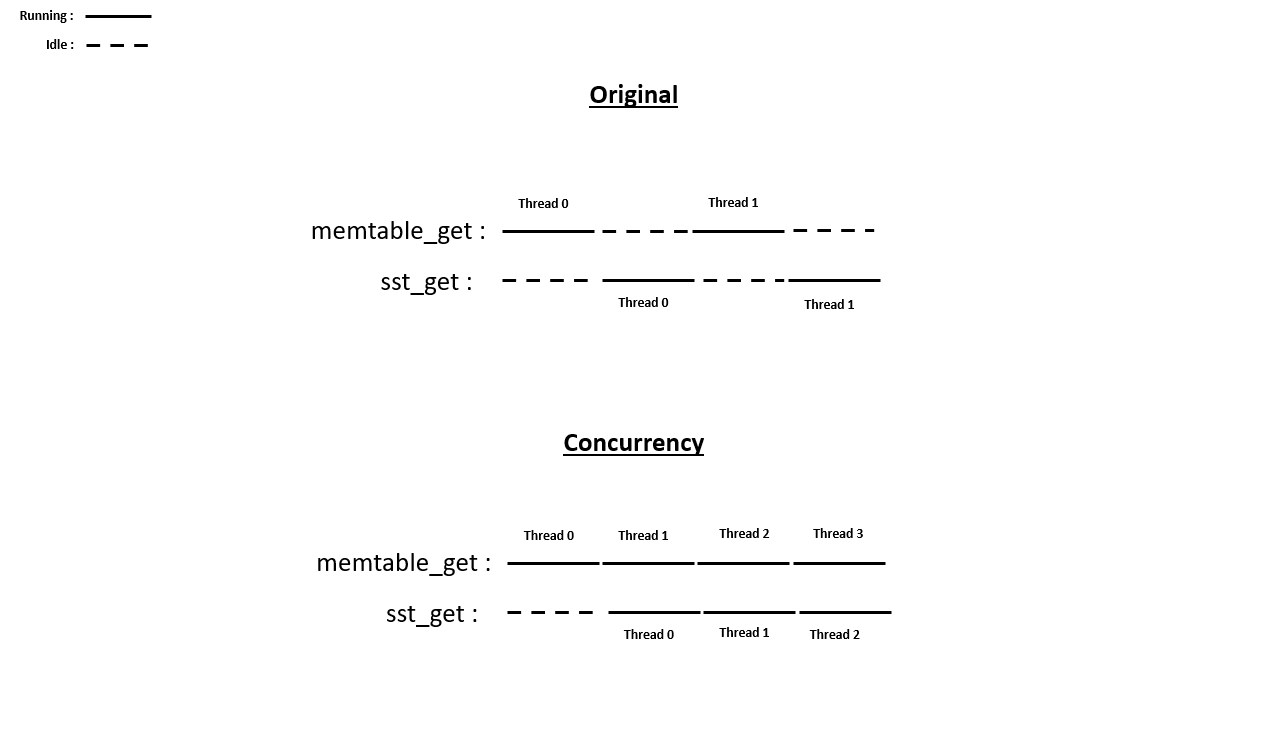
**pthread\_mutex\_unlock(&mymutex\_get\_memtable);**

**pthread\_mutex\_lock(&mymutex\_get\_sst);**

**return sst\_get(self->sst, key, value);**

*Παράδειγμα:* Έστω δύο threads t1 & t2 με t1 να περνά από το πρώτο lock. Καλεί την memtable\_get ψάχνοντας στην memtable για το κλειδί. Αν δεν το βρει κάνει unlock το mutex της memtable και κλειδώνει του sst. Παράλληλα, αφού το mutex της memtable είναι ξεκλείδωτο, το t2 νήμα επιτρέπεται να εκτελέσει την memtable\_get ενώ το t1 νήμα βρίσκεται στην sst\_get. Άρα εκτελείται παράλληλη αναζήτηση στις δύο δομές.

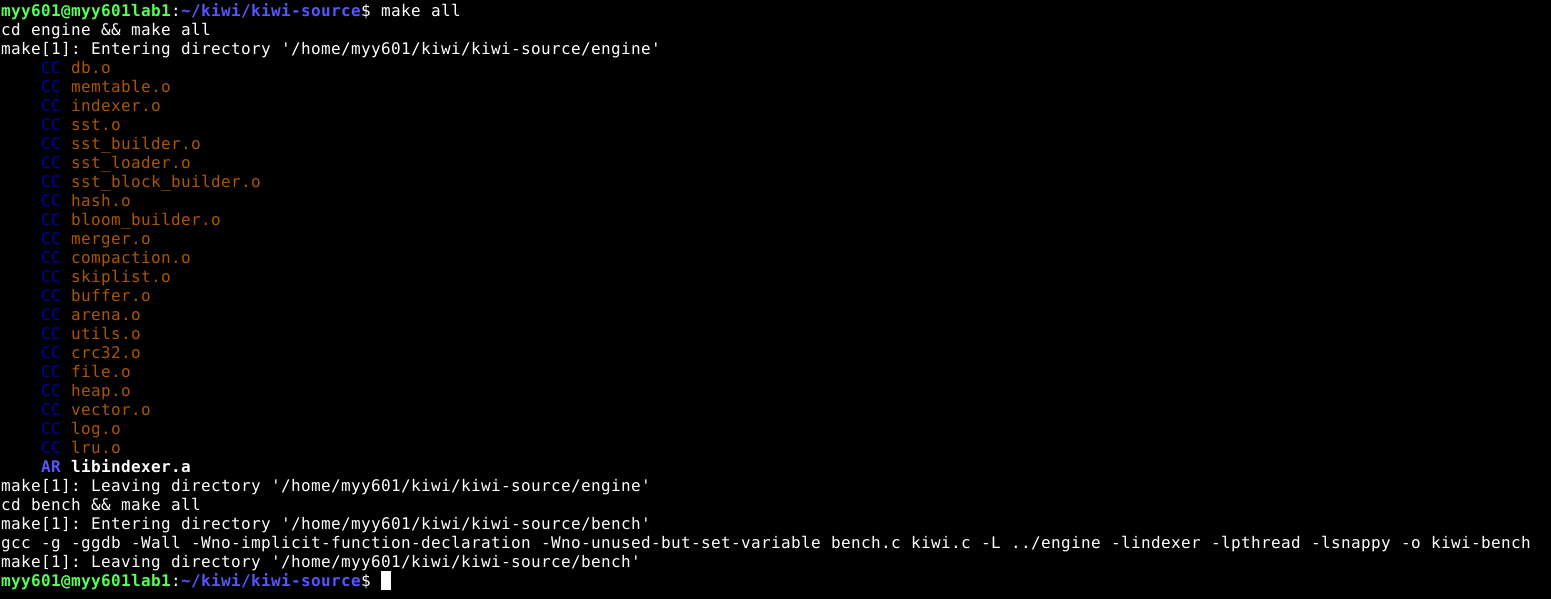
Παρακάτω φαίνεται μια εικονική περιγραφή της αρχικής σειριακής εκτέλεσης των memtable\_get και sst\_get σε αντίθεση με την δική μας υλοποίηση παραλληλισμού.



* Για την db\_add: Προσπαθήσαμε να εισάγουμε παραλληλισμό στην db\_add.

Παρ’ όλα αυτά δεν βρήκαμε δομές, που να μπορούν να εκτελεστούν παράλληλα, αφού οι εγγραφές γίνονται μόνο στην δομή memtable(μπορεί να εκτελεστεί ένα write την φορά).

*Εκτέλεση της εντολής make all στο φάκελο kiwi-source.*



*Παρακάτω παραθέτουμε παραδείγματα εκτέλεσης του προγράμματος και επεξηγήσεις αυτών.*

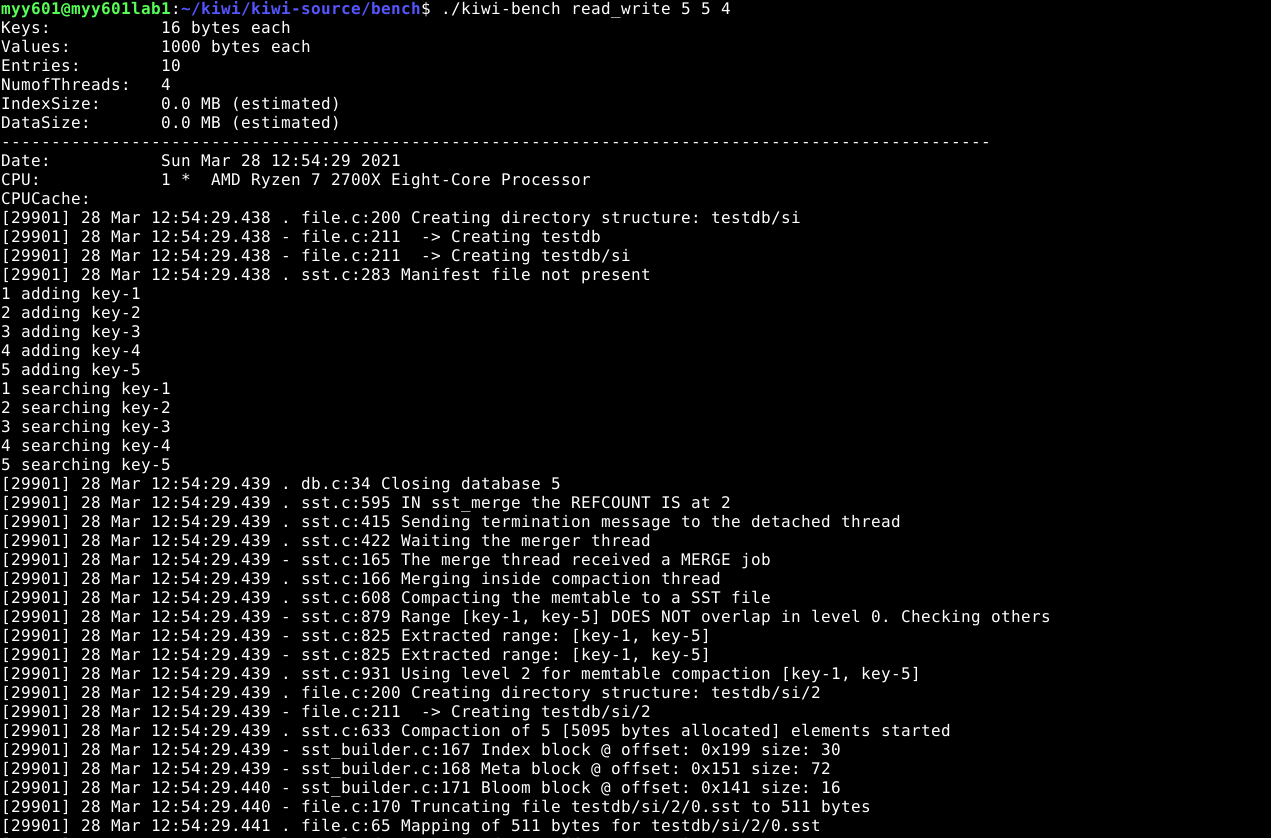
*Παράδειγμα 1:* Εκτελώντας το πρόγραμμα και δίνοντας στην γραμμή εντολών:

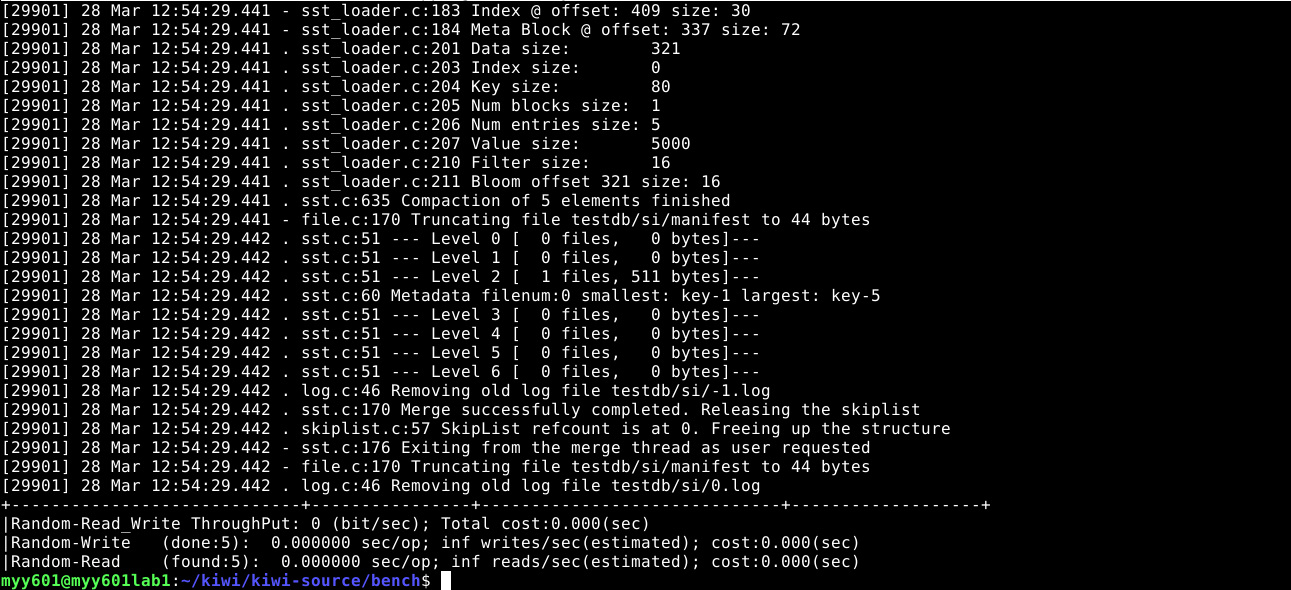
./kiwi-bench read\_write 5 5 4 (5 read, 5 write, 4 threads),

αρχικά εμφανίζονται πληροφορίες για το περιβάλλον του υπολογιστή και για το header. Το πρόγραμμα εκτελείται με 4 νήματα από τα οποία τα 2 αναθέτονται για write και 2 για read. Γίνονται πρώτα 5 εισαγωγές κλειδιών-τιμών και στην συνέχεια 5 αναζητήσεις κλειδιών-τιμών από τα οποία βρίσκει και τα 5.

Ο χρόνος εκτέλεσης είναι 0sec και για αυτό η ρυθμοαπόδοση είναι ίση με μηδέν (δεν ωφελεί να διαιρέσουμε τα bits με το μηδέν).

Στο τέλος εμφανίζονται πληροφορίες για την κατάσταση της βάσης (πχ τα compaction και τα merge καθώς και την κατάσταση των 7 επιπέδων του sst).





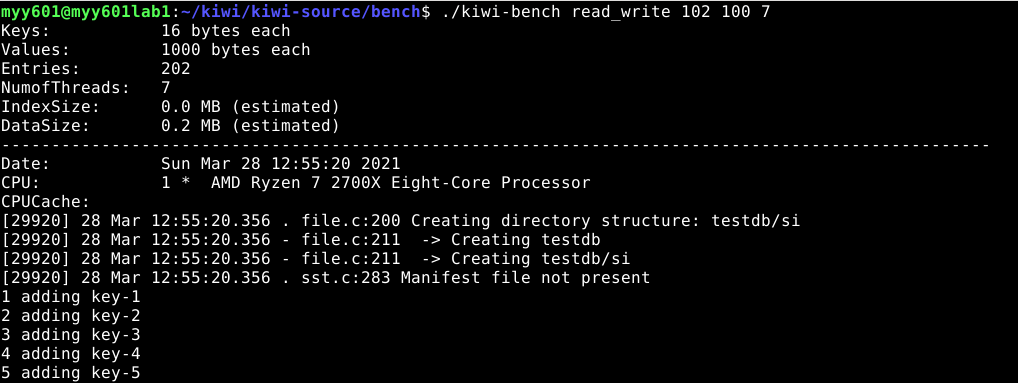
Εικόνα 1. Εκτέλεση ./kiwi-bench read\_write 5 5 4

*Παράδειγμα 2:* Εκτελώντας το πρόγραμμα και δίνοντας στην γραμμή εντολών:

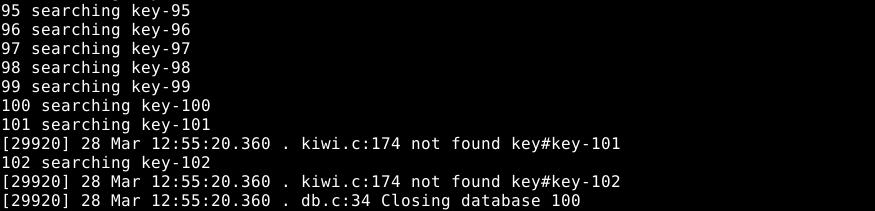
./kiwi-bench read\_write 102 100 7 (102 read, 100 write, 7 threads)

Εκτελούνται σωστά οι λειτουργίες (102)read και (100)write. Από τις 102 αναζητήσεις κλειδιών-τιμών βρίσκει μόνο τις 100 όπως φαίνεται στην Εικόνα 3.

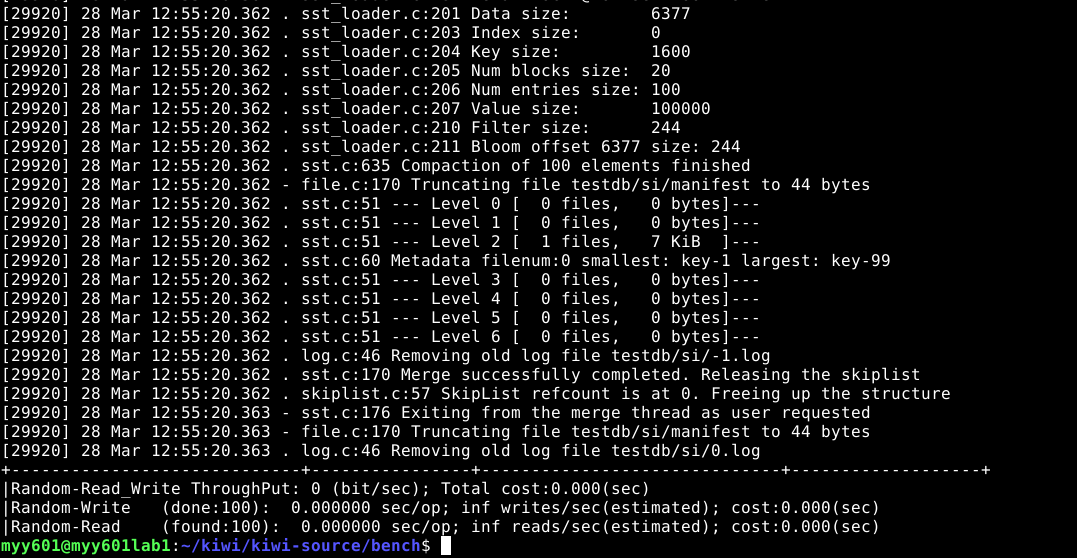
Στο τέλος της εκτέλεσης εμφανίζεται ο αριθμός των αιτήσεων που έγιναν για read και write καθώς και τον χρόνο εκτέλεσης που είναι ίσος με μηδέν. Γι’ αυτόν τον λόγο η ρυθμοαπόδοση είναι ίση με μηδέν.



Εικόνα 2. Εκτέλεση ./kiwi-bench read\_write 102 100 7



Εικόνα 3. Εμφάνιση κλειδιών 101 και 102

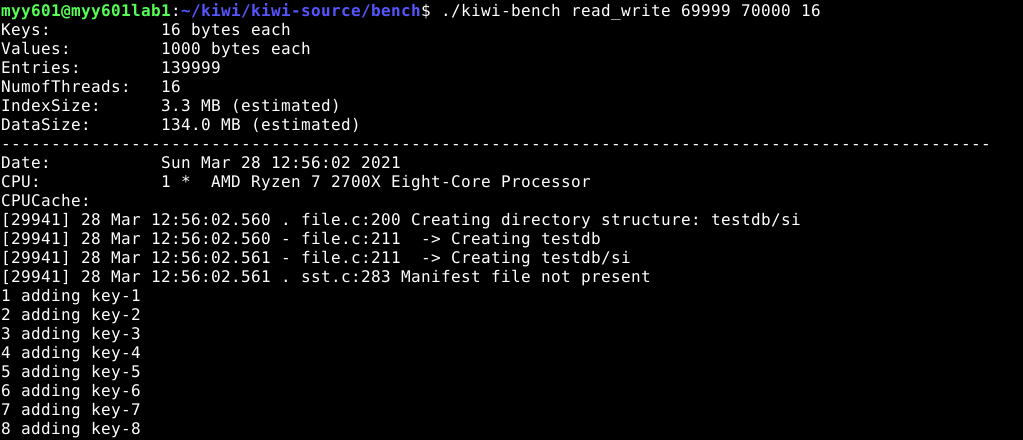


Εικόνα 4. Εμφάνιση του τέλους του προγράμματος

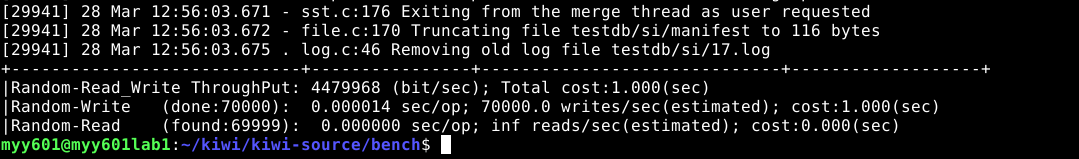
*Παράδειγμα 3:* Εκτελώντας το πρόγραμμα και δίνοντας στην γραμμή εντολών:

./kiwi-bench read\_write 69999 70000 16 (69999 read και 70000 write με 16 threads)

εκτελούνται 70000 εγγραφές κλειδιών-τιμών και 69999 αναζητήσεις κλειδιών-τιμών από τα οποία βρίσκει και τα 69999 (Εικόνα 5-6).



Εικόνα 5. Εκτέλεση ./kiwi-bench read\_write 69999 70000 16



Εικόνα 6. Εμφάνιση του τέλους εκτέλεσης του προγράμματος

ΣΤΑΤΙΣΤΙΚΑ

Για την εξαγωγή στατιστικών, θα υλοποιήσουμε την εκτέλεση του προγράμματος απαιτώντας 600.000 reads και 600.000 writes με τον αριθμό των νημάτων να είναι αρχικά 2 και στην συνέχεια 4,8,16,32,64,128.

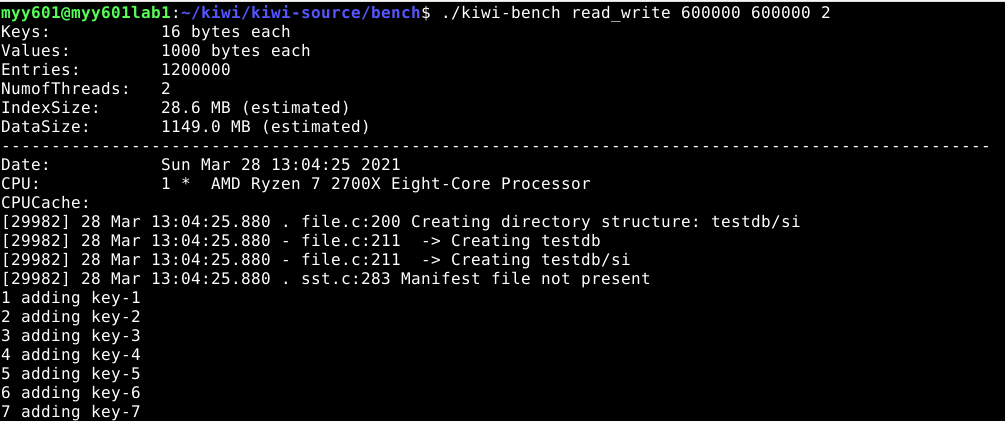
Εκτέλεση του προγράμματος με 2 νήματα:

Αρχικά εκτελώντας ./kiwi-bench read\_write 600000 600000 2

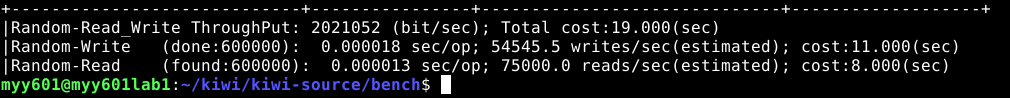
το πρόγραμμα δημιουργεί 600000 εγγραφές κλειδιών-τιμών και στην συνέχεια κάνει 600000 αναζητήσεις κλειδιών-τιμών όπου και τις βρίσκει όλες (όπως φαίνεται και στην Εικόνα 7-8).

Ο χρόνος απόκρισης είναι 19sec εκ των οποίων τα 11sec αφορούν τις εγγραφές και τα 8sec την αναζήτηση (Εικόνα 8).

Επίσης, η ρυθμοαπόδοση είναι ίση με 2021052 bit/sec.



Εικόνα 7. Εκτέλεση ./kiwi-bench read\_write 600000 600000 2

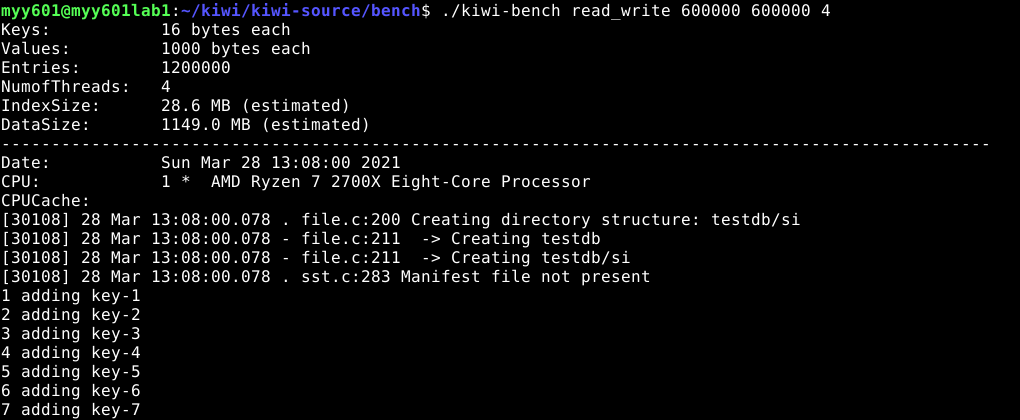


Εικόνα 8. Εμφάνιση του τέλους εκτέλεσης του προγράμματος

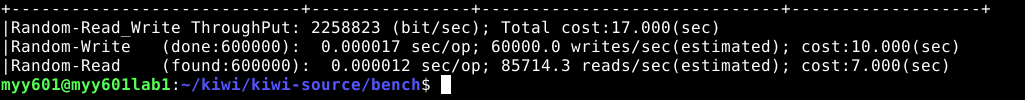
Εκτέλεση του προγράμματος με 4 νήματα:

Εκτελώντας το πρόγραμμα με 4 νήματα, το πρόγραμμα εκτελείται γρηγορότερα αφού ο χρόνος απόκρισης είναι ίσος με τα 17sec, 10 για τις εγγραφές και 7 για τις αναζητήσεις.

Ακόμη, η ρυθμοαπόδοση είναι ίση με 2258823 bit/sec, όπως είναι αναμενόμενο αφού αυξάνοντας τα νήματα περιμένουμε ότι ο όγκος των bits θα εγγράφεται και θα αναζητείται γρηγορότερα από ό,τι με 2 νήματα.



Εικόνα 9. Εκτέλεση ./kiwi-bench read\_write 600000 600000 4

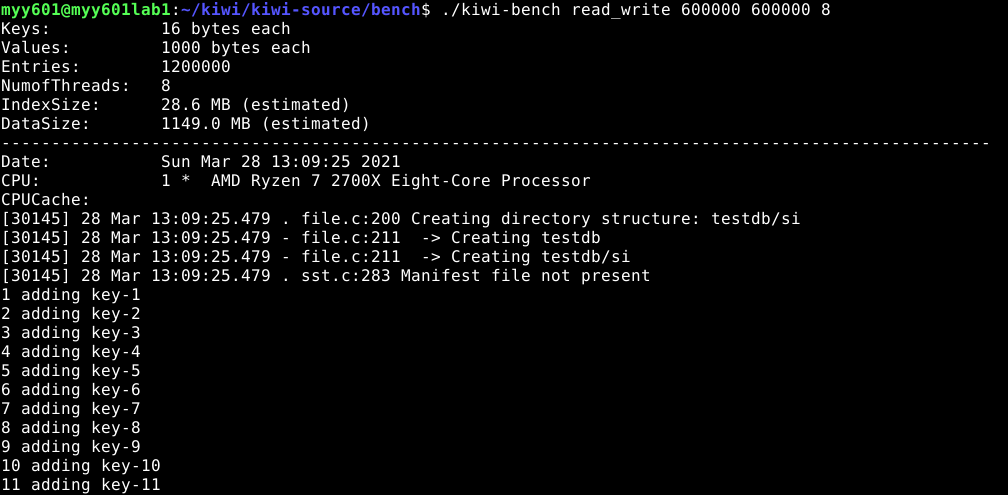


Εικόνα 10. Εμφάνιση του τέλους εκτέλεσης του προγράμματος

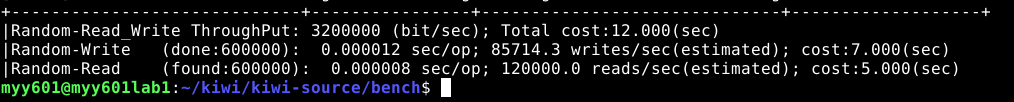
Εκτέλεση του προγράμματος με 8 νήματα:

Εκτελώντας το πρόγραμμα με 8 νήματα, το πρόγραμμα εκτελείται γρηγορότερα αφού ο χρόνος απόκρισης είναι ίσος με τα 12sec, 7 για τις εγγραφές και 5 για τις αναζητήσεις.

Ακόμη, η ρυθμοαπόδοση είναι ίση με 3200000 bit/sec, όπως είναι αναμενόμενο αφού αυξάνοντας τα νήματα περιμένουμε ότι ο όγκος των bits θα εγγράφεται και θα αναζητείται γρηγορότερα από ό,τι με λιγότερα νήματα.



Εικόνα 11. Εκτέλεση ./kiwi-bench read\_write 600000 600000 8

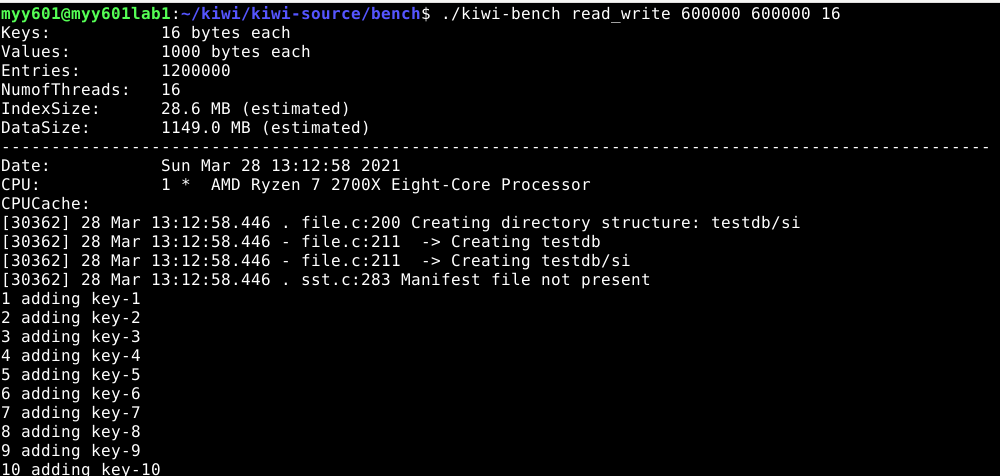


Εικόνα 12. Εμφάνιση του τέλους εκτέλεσης του προγράμματος

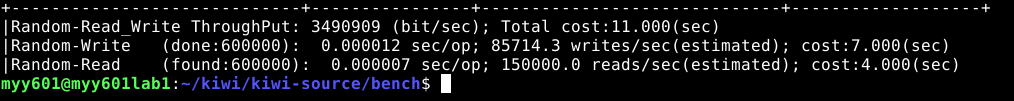
Εκτέλεση του προγράμματος με 16 νήματα:

Εκτελώντας το πρόγραμμα με 16 νήματα (Εικόνα 13), το πρόγραμμα εκτελείται γρηγορότερα αφού ο χρόνος απόκρισης είναι ίσος με τα 11sec, 7 για τις εγγραφές και 4 για τις αναζητήσεις (Εικόνα 14).

Ακόμη, η ρυθμοαπόδοση είναι ίση με 3490989 bit/sec, όπως είναι αναμενόμενο αφού αυξάνοντας τα νήματα περιμένουμε ότι ο όγκος των bits θα εγγράφεται και θα αναζητείται γρηγορότερα από ό,τι με λιγότερα νήματα.



Εικόνα 13. Εκτέλεση ./kiwi-bench read\_write 600000 600000 16

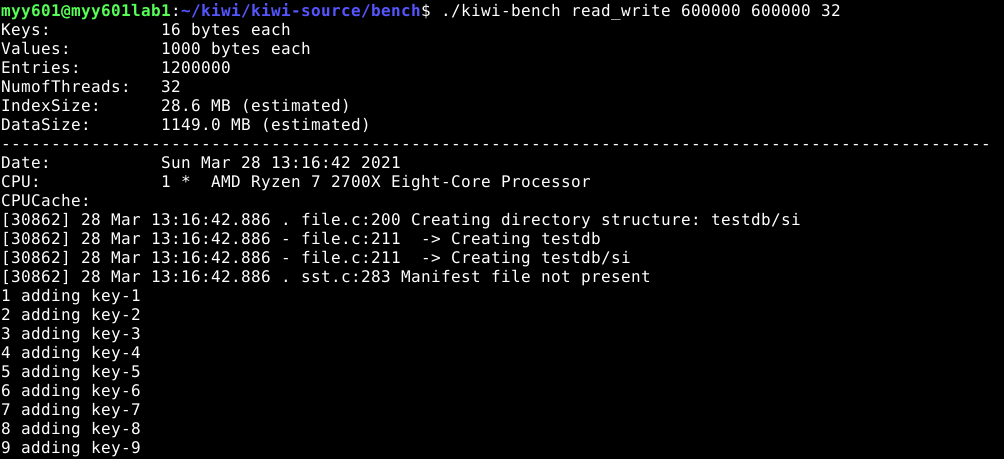


Εικόνα 14. Εμφάνιση του τέλους εκτέλεσης του προγράμματος

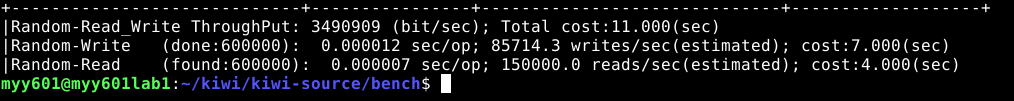
Εκτέλεση του προγράμματος με 32 νήματα:

Εκτελώντας το πρόγραμμα με 32 νήματα (Εικόνα 15), το πρόγραμμα εκτελείται γρηγορότερα αφού ο χρόνος απόκρισης είναι ίσος με τα 11sec, 7 για τις εγγραφές και 4 για τις αναζητήσεις (Εικόνα 16).

Ακόμη, η ρυθμοαπόδοση είναι ίση με 3490989 bit/sec, όπως είναι αναμενόμενο αφού αυξάνοντας τα νήματα περιμένουμε ότι ο όγκος των bits θα εγγράφεται και θα αναζητείται γρηγορότερα από ό,τι με λιγότερα νήματα.



Εικόνα 15. Εκτέλεση ./kiwi-bench read\_write 600000 600000 32

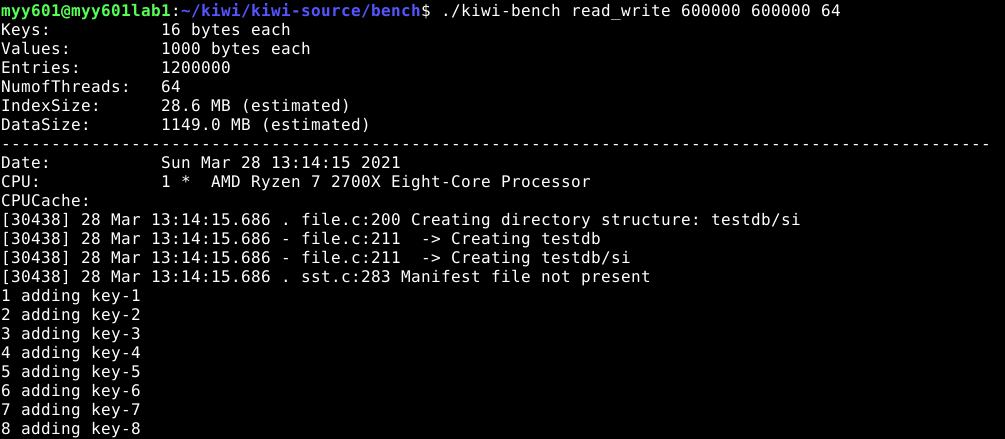


Εικόνα 16. Εμφάνιση του τέλους εκτέλεσης του προγράμματος

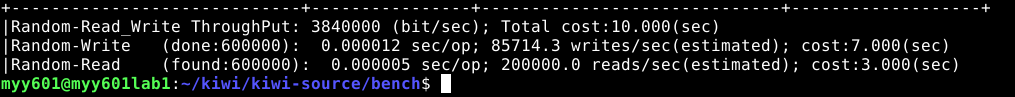
Εκτέλεση του προγράμματος με 64 νήματα:

Εκτελώντας το πρόγραμμα με 64 νήματα (Εικόνα 17), το πρόγραμμα εκτελείται γρηγορότερα αφού ο χρόνος απόκρισης είναι ίσος με τα 10sec, 7 για τις εγγραφές και 3 για τις αναζητήσεις (Εικόνα 18).

Ακόμη, η ρυθμοαπόδοση είναι ίση με 3840000 bit/sec, όπως είναι αναμενόμενο αφού αυξάνοντας τα νήματα περιμένουμε ότι ο όγκος των bits θα εγγράφεται και θα αναζητείται γρηγορότερα από ό,τι με λιγότερα νήματα.



Εικόνα 17. Εκτέλεση ./kiwi-bench read\_write 600000 600000 64

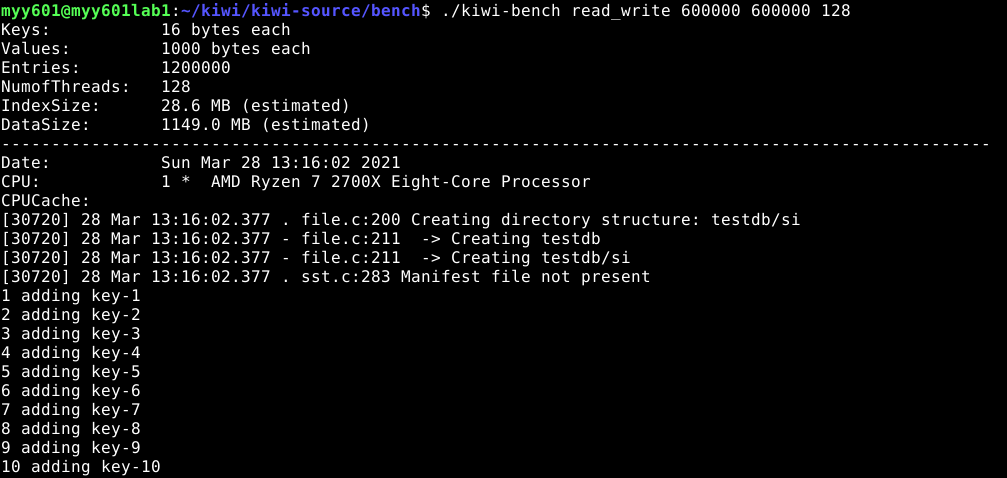


Εικόνα 18. Εμφάνιση του τέλους εκτέλεσης του προγράμματος

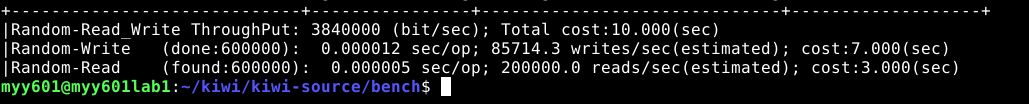
Εκτέλεση του προγράμματος με 128 νήματα:

Εκτελώντας το πρόγραμμα με 128 νήματα (Εικόνα 19), το πρόγραμμα εκτελείται γρηγορότερα αφού ο χρόνος απόκρισης είναι ίσος με τα 10sec, 7 για τις εγγραφές και 3 για τις αναζητήσεις (Εικόνα 20).

Ακόμη, η ρυθμοαπόδοση είναι ίση με 3840000 bit/sec, όπως είναι αναμενόμενο αφού αυξάνοντας τα νήματα περιμένουμε ότι ο όγκος των bits θα εγγράφεται και θα αναζητείται γρηγορότερα από ό,τι με λιγότερα νήματα.

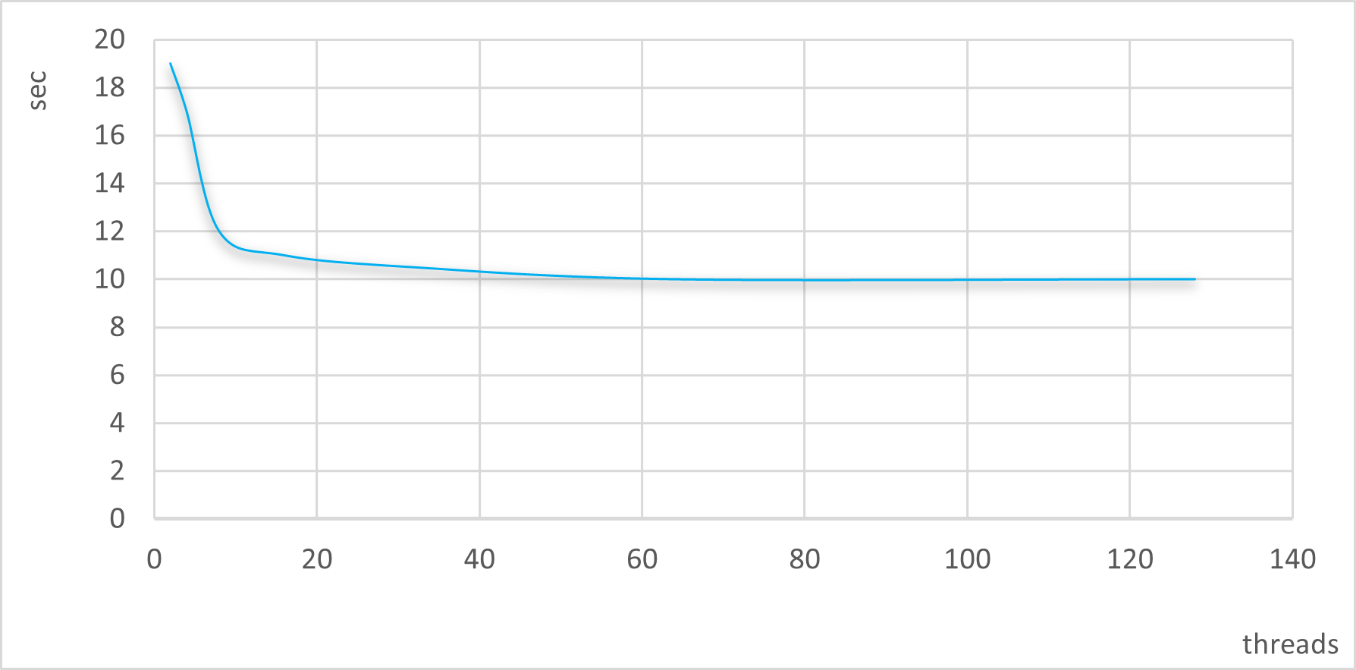


Εικόνα 19. Εκτέλεση ./kiwi-bench read\_write 600000 600000 128



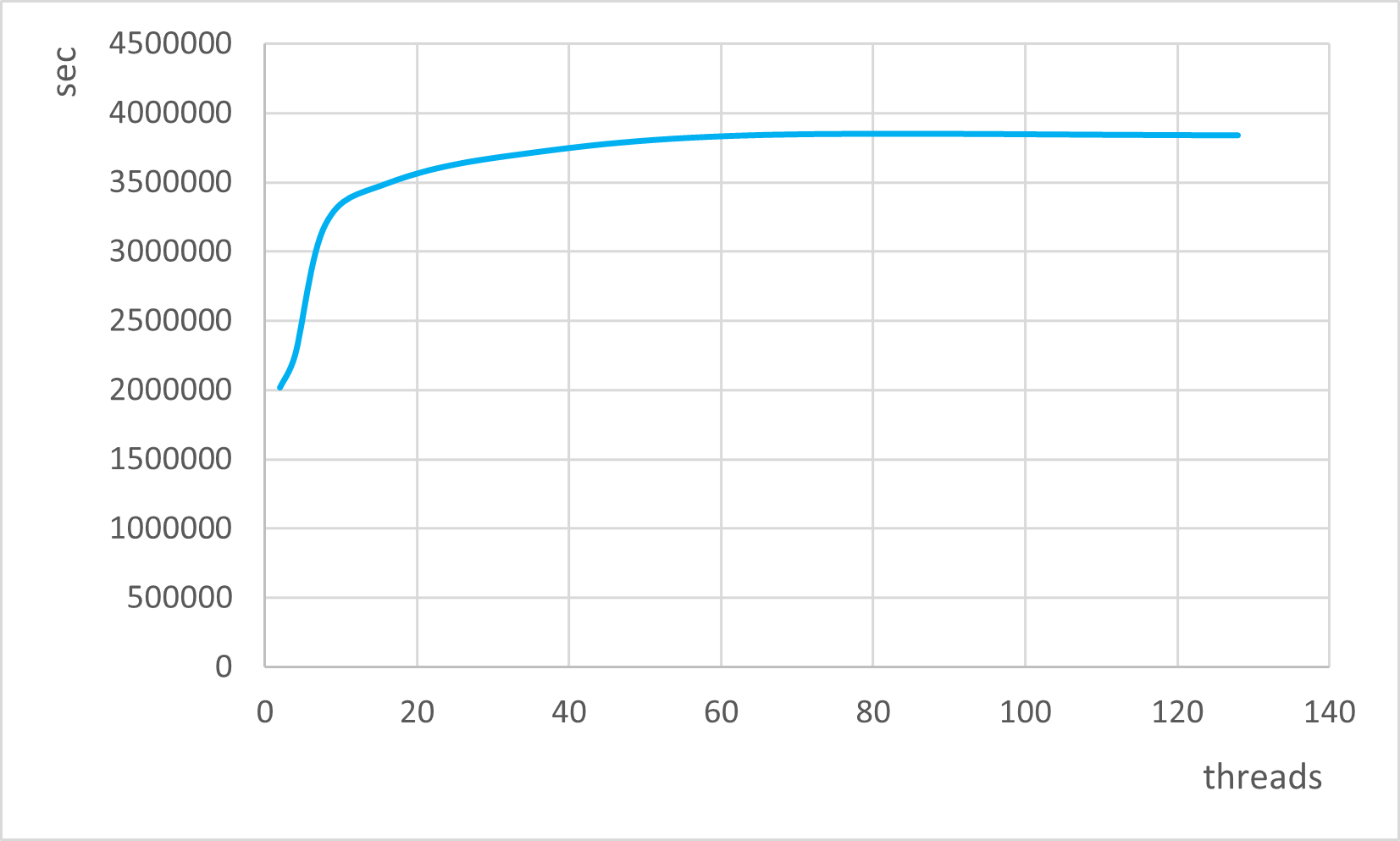
Εικόνα 20. Εμφάνιση του τέλους εκτέλεσης του προγράμματος

*Τα παραπάνω αποτελέσματα τα περνάμε σε excel και βγάζουμε τις παρακάτω γραφικές παραστάσεις.*



Εικόνα 21. Γραφική παράσταση για την απόδοση της read\_write

Όπως φαίνεται στην (Εικόνα 21), η απόδοση αυξάνεται παροδικά του χρόνου με την αύξηση των νημάτων. Για περισσότερα νήματα, ο χρόνος απόδοσης μειώνεται.



Εικόνα 21. Γραφική παράσταση για την απόδοση με βάση της ρυθμοαπόδοσης

Όπως φαίνεται από τα παραπάνω στατιστικά, με την αύξηση των νημάτων, το βέλτιστο κόστος είναι 10sec με 64 νήματα, και μετά αυξάνοντας τα νήματα δεν υπάρχει άλλη βελτίωση.

* *Όρια εκτέλεσης του προγράμματος:*

Για λιγότερα από 100.000 read&write με 2 νήματα, το κόστος θα είναι μηδέν.   
Για περισσότερα από 4 εκατομμύρια read&write με 2 νήματα, θα δημιουργηθεί bus error, όπως είναι αναμενόμενο αφού η βάση έχει όριο χωρητικότητας.