ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ

1Η ΕΡΓΑΣΤΗΡΙΑΚΗ ΑΣΚΗΣΗ

Μάρτιος 2022

Υλοποίηση πολυνηματικής λειτουργίας σε μηχανή αποθήκευσης δεδομένων

Ομάδα :

Αθανασία-Δανάη Τσαούση 3349

Πηνελόπη Ελευθεριάδη 3221

ΠΕΡΙΕΧΟΜΕΝΑ

Στόχος Εργασίας

Η υλοποίηση της πολυνηματικής λειτουργίας των εντολών put και get σε μια μηχανή αποθήκευσης.

(1)Πρώτο Στάδιο

Μέσω αναζήτησης του κώδικα και με χρήση του εργαλείου gdb ,κατανοήσαμε την ροή του κώδικα και την σύνδεση των βασικών συναρτήσεων της μηχανής για τις λειτουργίες put και get.

Πιο συγκεκριμένα :

- Στο αρχείο **bench**

-bench.c: Περιέχει την main όπου ανάλογα με ποια λειτουργία ζητά ο χρήστης(write ή read) καλεί τις αντίστοιχες συναρτήσεις όπου βρίσκονται στο kiwi.c

-kiwi.c: Περιέχει τις συναρτήσεις _write_test και _read_test, οι οποίες ανοίγουν την βάση(db_open) και την κλείνουν(db_close) και προσθέτουν το ζεύγος key-value ,μέσω της db_add(αντίστοιχα και για την λειτουργία read με την db_get).Επίσης, και στις 2 συναρτήσεις υπολογίζονται τα στατιστικά και εκτυπώνονται.

-bench.h: αρχείο κεφαλίδας όπου περιέχει κοινές μεταβλητές για τα αρχεία bench.c και kiwi.c.

- Στο αρχείο **engine**:

-db.c : Περιέχει τις db_add και db_get. Η db_add πρώτα ελέγχει αν το memtable χρειάζεται merge και αν ναι καλεί την sst_merge . Μετά κάνει εγγραφή με την memtable_add. Η db_get ψάχνει στην μνήμη με την memtable_get και άμα
δεν βρεθεί τότε ψάχνει στον δίσκο με την sst_get().

-db.h : αρχείο κεφαλίδας για την σύνδεση του engine με τα αρχεία του bench.

(2)Δεύτερο Στάδιο

(Α)Αλλαγές ώστε να ξεκινήσουμε πολλαπλά νήματα(ερώτημα 3)

Σε αυτό το στάδιο θα παρουσιάσουμε τις αλλαγές στο bench.c και στο bench.h για να μοιραστούν οι λειτουργίες write και read σε πολλαπλά νήματα.

Θα τροποποιηθεί ο τρόπος εκτέλεσης και από εδώ και πέρα θα τρέχουμε στην γραμμή εντολών με μορφή :

- ./kiwi-bench write 100 2

  Στην θέση 1, δηλαδή εκεί που έχουμε σε αυτό το παράδειγμα εκτέλεσης το write δίνουμε την λειτουργία που θέλουμε(write,read και αργότερα addget).

  Στην θέση 2(εδώ 100) δίνουμε πόσα ζεύγη key-value θα διαβαστούν ή θα γραφτούν(requests).

  Στην θέση 3(εδώ 2) δίνουμε πόσα νήματα θα εκτελέσουν την λειτουργία.

Για αυτό το στάδιο προσθέσαμε :

-Στο **bench.h** την δόμη args η οποία περιέχει τα ορίσματα της _write_test και _read_test, όπως και τον αριθμό των νημάτων.

```
struct args
{   int rArg;
long int countArg;
int numthr;
};
```

-Στο **bench.c** :

Προσθέσαμε τις συναρτήσεις

-void *callWrite(void * arg) : Δέχεται σαν όρισμα την δομή args και σκοπός της είναι η κλήση της _write_test με τα κατάλληλα ορίσματα (count,r), αφού τα εξάγει από την δομή.

Πριν την κλήση της _write_test θα πρέπει να δοθεί το κατάλληλο count,το οποίο επιτυγχάνουμε με την splitCountToThreads.

-void *callRead(void *arg):Αντίστοιχα για την λειτουργία read, η συνάρτηση αυτή δέχεται σαν όρισμα την δομή args και σκοπός της είναι η κλήση της _read_test με τα κατάλληλα ορίσματα (count,r), αφού τα εξάγει από την δομή. Πριν την κλήση της _read_test θα πρέπει να δοθεί το κατάλληλο count,το οποίο επιτυγχάνουμε με την splitCountToThreads.

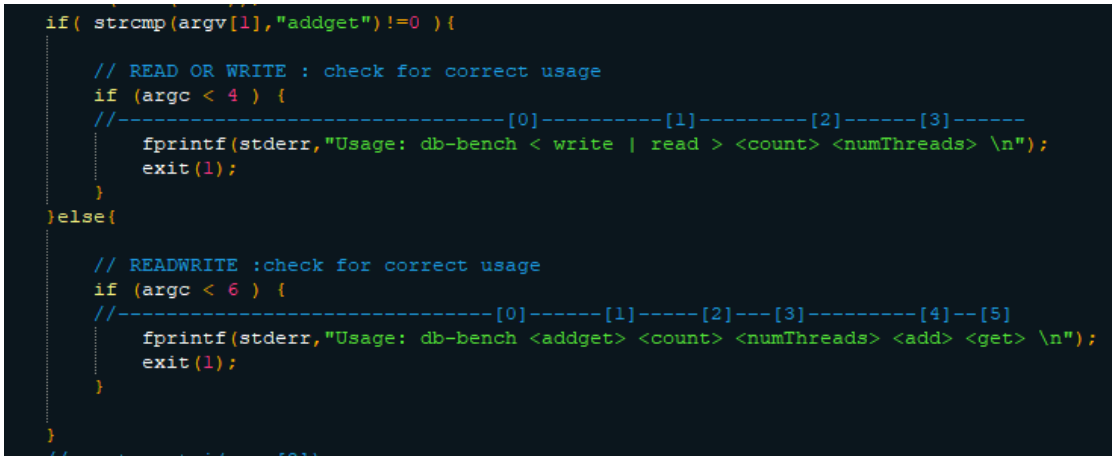
- long int splitCountToThreads(long int count,int numthreads): Στόχος της είναι να μοιράσει το count ,δηλαδή πόσα ζεύγη θα πρέπει να γραφτούν ή διαβαστούν, σε κάθε νήμα. Πρακτικά, χωρίζει την δουλειά που πρέπει να γίνει ανά τα νήματα.

Αλλαγές στην main :

-Προσθέσαμε τις **μεταβλητές** *struct args strArgs*, που είναι η δομή που θα χρησιμοποιηθεί από τις λειτουργίες, και την *int numthreads* , που είναι ο αριθμός των threads που δίνει ο χρήστης (*numthreads = atoi(argv[3])*).

-Έλεγχος

Αφού ο χρήστης πληκτρολογεί την εντολή για την εκάστοτε λειτουργία ,εμείς ελέγχουμε τη σωστή χρήση της. Αν δεν πρόκειται για τη λειτουργία addget που έχει παραπάνω ορίσματα ,ο χρήστης πρέπει να δώσει 4 ορίσματα είτε για write είτε για read .Αν τα ορίσματα είναι λιγότερα από 4 τότε κάνουμε έξοδο και ενημερώνουμε το χρήστη με εκτύπωση της σωστής χρήσης της εντολής. Οπότε στο else ελέγχουμε τον αριθμό ορισμάτων αντίστοιχα για την εντολή addget στην οποία τα ορίσματα είναι 6.Αν είναι λιγότερα από 6 εκτυπώνουμε και πάλι τη σωστή χρήση της εντολής.



-Μετά , ανάλογα τη λειτουργία που ζητά ο χρήστης μπαίνει ο κώδικας στην αντίστοιχη if.

Για την λειτουργία write :

Κάνουμε την αλλαγή **if (argc == 5)** λόγω αύξησης των ορισμάτων ,δηλαδή η προσθήκη του αριθμού των νημάτων. Λαμβάνουμε την τιμή από την γραμμή εντολών με τον παρακάτω τρόπο

numthreads = atoi(argv[3]);

Και στη συνέχεια δημιουργούμε έναν πίνακα με τα id των νημάτων (**pthread_t thrIds[numthreads]**)

Καλούμε την **controlDb(1)** για να ανοίξουμε την αποθήκη. Μετά, γεμίζουμε το struct, με το **r**, το **count** και το **numthreads**(ο αριθμός νημάτων δηλαδή).Τα δύο τελευταία τα λαμβάνουμε από τον χρήστη όπως αναλύσαμε παραπάνω.

-Μετά, με 2 for δημιουργούμε τα νήματα και περιμένουμε να τελειώσει κάθε νήμα τις λειτουργίες του.

Σε αυτό το σημείο, κάθε νήμα εκτελεί την **callWrite ,**που θα καλέσει με τα σωστά ορίσματα την _write_test.

Τέλος, κλείνουμε την αποθήκη *controlDb(0).*

```
if (strcmp(argv[1], "write") == 0) {

        int r = 0;
        ////assign given num of requests to count Var
        count = atoi(argv[2]);

        //prints
        _print_header(count);
        _print_environment();

        //argc = 5 - > 0..4
        if (argc == 5)
                r = 1;

        //synolika threads apo ton xristi
        numthreads = atoi(argv[3]);

        pthread_t thrIds[numthreads];
        controlDb(1);

        strArgs.rArg=r;
        strArgs.countArg=count;
        strArgs.numthr=numthreads;
        for(i=0;i<numthreads;i++)
                pthread_create(&thrIds[i],NULL,callWrite,(void *)&strArgs);
        for(i=0;i<numthreads;i++)
                pthread_join(thrIds[i],NULL);

        controlDb(0);


-       performanceStats(count,wrCost,1);
```

Για την λειτουργία read :

Κάνουμε την αλλαγή **if (argc == 5)** λόγω αύξησης των ορισμάτων ,δηλαδή η προσθήκη του αριθμού των νημάτων. Λαμβάνουμε την τιμή από την γραμμή εντολών με τον παρακάτω τρόπο

numthreads = atoi(argv[3]);

Και στη συνέχεια δημιουργούμε έναν πίνακα με τα id των νημάτων (**pthread_t thrIds[numthreads]**)

Καλούμε την **controlDb(1)** για να ανοίξουμε την αποθήκη. Μετά, γεμίζουμε το struct, με το **r**, το **count** και το **numthreads**(ο αριθμός νημάτων δηλαδή).Τα δύο τελευταία τα λαμβάνουμε από τον χρήστη όπως αναλύσαμε παραπάνω.

-Μετά, με 2 for δημιουργούμε τα νήματα και περιμένουμε να τελειώσει κάθε νήμα τις λειτουργίες του.

Σε αυτό το σημείο, κάθε νήμα εκτελεί την **callRead,**που θα καλέσει με τα σωστά ορίσματα την _read_test.

Τέλος, κλείνουμε την αποθήκη *controlDb(0).*

```
} else if (strcmp(argv[1], "read") == 0) {

        int r = 0;
        //assign given num of requests to count Var
        count = atoi(argv[2]);

        //prints
        _print_header(count);
        _print_environment();

        //argc=5 ->0..4
        if (argc == 5)
                r = 1;

        //synolika threads apo ton xristi
        numthreads = atoi(argv[3]);
        pthread_t thrIds[numthreads];
        controlDb(1);

        strArgs.rArg=r;
        strArgs.countArg=count;
        strArgs.numthr=numthreads;
        for(i=0;i<numthreads;i++)
                pthread_create(&thrIds[i],NULL,callRead,(void *)&strArgs);
        for(i=0;i<numthreads;i++)
                pthread_join(thrIds[i],NULL);

        controlDb(0);


        performanceStats(count,rdCost,2);
```

Αλλαγές στο αρχείο kiwi.c:

-Εδώ αρχικά έχουμε την DB* db ως global μεταβλητή πλέον. Αυτό το κάναμε διότι πλέον, λόγω της χρήσης νημάτων δεν μπορούμε να ανοίγουμε και να κλείνουμε την μηχανή μέσα στις συναρτήσεις _write_test και _read_test, γιατί καλούνται από κάθε νήμα ξεχωριστά. Επίσης, έχουμε δημιουργήσει μία νέα συνάρτηση , την controlDb(int flagOper), η οποία είναι υπεύθυνη για το άνοιγμα και το κλείσιμο της βάσης. Με την χρήση ενός flag ,του flagOper, η bench.c ανοίγει , με το όρισμα flagOper ίσο με 1, την μηχανή και αντίστοιχα με flagOper ίσο με το 0 την κλείνει.

```
void controlDb(int flagOper){
        if(flagOper == 1){
                db = db_open(DATAS);
        }else if(flagOper == 0){
                db_close(db);
        }
}
```

Μέσα στις υπάρχουσες συναρτήσεις _write_test και _read_test κάναμε τις εξής αλλαγές :

(1) Αφαιρέσαμε την μεταβλητή cost, αφού πλέον το επεξεργαζόμαστε στην computeCost.

(2) Αφαιρέσαμε τις λειτουργίες db_open και db_close,αφού τις μεταφέραμε στο controlDb.

(3) Τέλος, αφαιρέσαμε τις εκτυπώσεις των στατιστικών, οι οποίες πλέον επεξεργάζονται από την computeCost και εκτυπώνονται στο bench.c στην performanceStats. Αυτά, όμως, τα αναλύουμε παρακάτω.

(Β)Πρόσθετη λειτουργία get/put (ερώτημα 5)

Μας ζητείται η προσθήκη μίας λειτουργίας όπου εκτελεί το μίγμα λειτουργιών put και get. Ο τρόπος που το προσεγγίσαμε είναι ο εξής:

-Αρχικά, προσθέσαμε στην γραμμή εντολών δύο ακόμα ορίσματα , των αριθμό των νημάτων για την λειτουργία write και των αριθμό των νημάτων για την λειτουργία read.

- Παράδειγμα εκτέλεσης : ./kiwi-bench addget 250000 100 50 50

```
myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench addget 250000 100 50 50
```

Προφανώς, όπως και στις προηγούμενες λειτουργίες, στη θέση 3 δίνουμε πόσα νήματα θα έχουμε συνολικά και για τις δύο λειτουργίες.

-Ορίζουμε κάποιες νέες μεταβλητές, οι οποίες είναι οι wrperc, rdperc που θα δώσουμε το ποσοστό για κάθε λειτουργία και οι writeCount, readCount που είναι το ποσοστό του count για κάθε λειτουργία.

- Λαμβάνουμε το σύνολο των νημάτων, τα νήματα για την λειτουργία write και τα νήματα για την λειτουργία read με τον εξής τρόπο :

```
numthreads = atoi(argv[3]);

int wrnumthreads = atoi(argv[4]);

int rdnumthreads = atoi(argv[5]);
```

- Κάνουμε την αλλαγή **if (argc == 7)** λόγω αύξησης των ορισμάτων.

-Καλούμε την **controlDb(1)** για να ανοίξουμε την αποθήκη. Μετά, υπολογίζουμε τα ποσοστά για την κάθε λειτουργία και τα χρησιμοποιούμε για να βρούμε το count για τις λειτουργίες.

Αρχικά , υπολογίζουμε το ποσοστό με χρήση των αριθμό των νημάτων για την κάθε λειτουργία και των συνολικών νημάτων που έχουμε.

wrperc= (wrnumthreads * 100 / numthreads);

rdperc= (rdnumthreads * 100 / numthreads);

Μετά, βρίσκουμε το count της κάθε λειτουργίας και το μετατρέπουμε από ποσοστό σε long int.

writeCount = (long int) (count*wrperc / 100);

readCount = (long int) (count*rdperc / 100);

-Έπειτα, καλούμε την συνάρτηση ***createThreadsforAddGet****(wrnumthreads,rdnumthreads,writeCount,readCount, r,wrperc,rdperc),*με ορίσματα τον αριθμό των threads για την write, μετά για την read , τα ποσοστά για το count και τα ποσοστά wrperc, rdperc.

-Επιστρέφοντας , κλείνουμε την αποθήκη *controlDb(0).*

Επίσης, θα περιγράψουμε την λειτουργία της ***createThreadsforAddGet.***

--createThreadsforAddGet(int wrthreads,int rdthreads,long int writeCount,long int readCount,int r,int wrperc,int rdperc)

Ο ρόλος αυτής της συνάρτησης είναι η δημιουργία των νημάτων για κάθε λειτουργία με τα σωστά ορίσματα, σύμφωνα με τα ποσοστά. Η εύρεση των ποσοστών έχει γίνει στην main , όπως βλέπουμε παραπάνω , και βασίζεται στον αριθμό τον threads που έχει δώσει ο χρήστης για κάθε λειτουργία(wrnumthreads, numthreads).

-Δημιουργούμε έναν πίνακα με τα id των νημάτων για κάθε λειτουργία .Λόγω δυσκολιών και με σκοπό την απλοποίηση και την καλύτερη απόδοση του αλγορίθμου μας , θέτουμε 1500 αναγνωριστικά .

pthread_t addId[1500];

pthread_t getId[1500];

και φτιάχνουμε και τα αντίστοιχα structures.

Τέλος, για τις δύο λειτουργίες φτιάχνουμε τα νήματα που εκτελούν τις συναρτήσεις

callWrite και **callRead.** Περιμένουμε ,επίσης, να τελειώσει κάθε νήμα τις λειτουργίες του και αποδεσμέυουμε τους πόρους του με τη χρήση join.

```
} else if (strcmp(argv[1], "addget") == 0) {
        int wrperc = 0;//pososto gia write
        int rdperc = 0;//pososto gia write

        long int writeCount;
        long int readCount;

        //given num of threads for each operation
        int wrnumthreads = atoi(argv[4]);
        int rdnumthreads = atoi(argv[5]);

        int r = 0;
        //assign given num of requests to count Var
        count = atoi(argv[2]);

        //prints
        _print_header(count);
        _print_environment();


        //argc=7 -> 0..6
        if (argc == 7)
                r = 1;

        //synolika threads apo ton xristi
        numthreads = atoi(argv[3]);


        controlDb(1);//anoigma mixanis

        wrperc= (wrnumthreads * 100 / numthreads);
        rdperc= (rdnumthreads * 100 / numthreads);



        //pososto count gia kathe leitoyrgia
        writeCount = (long int) (count*wrperc / 100); //synolika write poy ua ginoyn
        readCount = (long int) (count*rdperc / 100); //synolika read poy ua ginoyn


        createThreadsforAddGet(wrnumthreads,rdnumthreads,writeCount,readCount,r,wrperc,rdperc);

        controlDb(0);

        performanceStats(writeCount,wrCost,1);
        performanceStats(readCount,rdCost,2);
```

```
void createThreadsforAddGet(int wrthreads,int rdthreads,long int writeCount,long int readCount,int r,int wrperc,int rdperc){


        //create tids for threads
        pthread_t addId[1500];
        pthread_t getId[1500];

        //ftiaxnoume structures (san enimerosh) gia write kai gia read antistoixa
        struct args writeargs;
        struct args readargs;

        writeargs.rArg=r;
        writeargs.countArg=writeCount;
        writeargs.numthr=wrthreads;

        readargs.rArg=r;
        readargs.countArg=readCount;
        readargs.numthr= rdthreads;

        //create threads and joins for write and read
        for(int i=0; i<wrthreads; i++){
                pthread_create(&addId[i],NULL,callWrite,(void*) &writeargs);
        }

        for(int i=0; i<rdthreads; i++){
                pthread_create(&getId[i],NULL,callRead,(void*) &readargs);
        }
        for(int i=0; i<wrthreads; i++){
                pthread_join(getId[i],NULL);
        }
        for(int i=0; i<rdthreads; i++){
                pthread_join(addId[i],NULL);
        }
```

Γ) Στατιστικά απόδοσης των λειτουργιών (ερώτημα 6)

Για τον αμοιβαίο αποκλεισμό μεταξύ των διαφορετικών νημάτων χρησιμοποιήσαμε 2 mutex ("mutual exclusion" : αμοιβαίος αποκλεισμός).Τα ονομάσαμε writeCmut , readCmut και προστατεύουν τη πρόσβαση σε ένα διαμοιραζόμενο πόρο (θέση μνήμης) που στην περίπτωση μας είναι οι global μεταβλητές wrCost, rdCost που είναι τα συνολικά κόστη για τις λειτουργίες write και read αντίστοιχα.

Τα mutex και τα κόστη τα δηλώνουμε στο bench.h που περιέχεται στο kiwi.c:

- double rdCost,wrCost;

- pthread_mutex_t readCmut;
- pthread_mutex_t writeCmut;

Στο **kiwi.c** φτιάξαμε μια συνάρτηση (computeCost) η οποία μπορεί να χρησιμοποιηθεί και για τις 2 λειτουργίες read και write με ανάλογη τροποποίηση στα ορίσματα που της δίνονται. Από τα ορίσματα long int firstTime,long int lastTime υπολογίζει την διαφορά τελικου μείον αρχικού χρόνου και τον περνάει στην προσωρινή μεταβλητη addTime.Εφόσον έχει υπολογιστεί η addTime ξεκινά η κρίσιμη περιοχή οπότε χρησιμοποιούμε τη ρουτίνα pthread_mutex_lock(&myMut) με το όρισμα της mutex που δίνεται(κλείδωμα myMut). Η addTime έπειτα προστίθεται στην μεταβλητή res (rdCost/wrCost) κι έτσι ενημερώνεται η τιμής της διαμοιραζόμενης μεταβλητής. Μετά την ανανέωση τελειώνει η κρίσιμη περιοχή οπότε χρησιμοποιούμε τη ρουτίνα pthread_mutex_unlock(&myMut) για να ξεκλειδώσουμε την myMut.Τέλος επιστρέφει την τιμή της ανανεωμένης μεταβλητής res.

```
double computeCost(long long firstTime,long long lastTime,double res,pthread_mutex_t myMut){
    long long addTime;
    //diafora telikou meion arxikou xronou
    addTime= lastTime-firstTime;
    pthread_mutex_lock(&myMut);
    //prosthesi tis diaforas
    res= res + addTime;
    pthread_mutex_unlock(&myMut);
    return res;

}
```

Η κλήση της computeCost γίνεται στο τέλος κάθε λειτουργίας (_write_test /read_test_) αφού έχουμε στη διάθεση μας και το χρόνο τερματισμού της λειτουργίας (end = get_ustime_sec();) και η επιστρεφόμενη τιμη αποθηκεύεται αντίστοιχα στις global wrCost, rdCost).

void _write_test(long int count, int r)

{

.

.

end = get_ustime_sec();

wrCost = computeCost(start,end,wrCost,writeCmut);

}

void _read_test (long int count, int r)

{

.

.

end = get_ustime_sec();

rdCost = computeCost(start,end,rdCost,readCmut);

}

Στο **bench.c** έχουμε φτιάξει μια συνάρτηση με όνομα performanceStats ,η οποία είναι υπεύθυνη να εκτυπώνει τα στατιστικά απόδοσης της μηχανής αποθήκευσης δεδομένων. Δέχεται 3 ορίσματα τον αριθμό των requests (count), το κόστος της λειτουργίας operCost και το είδος της λειτουργίας σε flagOper (read : flagOper=2, write :flagOper=1)

```
//print statistics of the performance
//when flag =1  -> write operation
//when flag =2  -> read operation
void performanceStats(long int count ,double operCost,int flagOper){
    printf(LINE);
    printf(LINE1);
    if(flagOper==1){
        printf("|Random-WRITE    (done:%ld): %.6f sec/op; %.1f writes/sec(estimated); cost:%.3f(sec);\n"
            ,count,(double)(operCost / count)
            ,(double)(count/ operCost)
            ,operCost);
    }else if(flagOper==2){
        printf("|Random-READ     (done:%ld): %.6f sec/op; %.1f reads/sec(estimated); cost:%.3f(sec);\n"
            ,count,(double)(operCost / count)
            ,(double)(count/ operCost)
            ,operCost);
    }


}
```

Αυτή την συνάρτηση την καλούμε με τα αντίστοιχα ορίσματα μετά την ολοκλήρωση κάθε λειτουργίας, όπως φαίνεται και στις παρακάτω εικόνες.

```
if (strcmp(argv[1], "write") == 0) {

    int r = 0;
    ////assign given num of requests to count Var
    count = atoi(argv[2]);

    //prints
    _print_header(count);
    _print_environment();

    //argc = 5 - > 0..4
    if (argc == 5)
        r = 1;

    //synolika threads apo ton xristi
    numthreads = atoi(argv[3]);

    pthread_t thrIds[numthreads];
    controlDb(1);

    strArgs.rArg=r;
    strArgs.countArg=count;
    strArgs.numthr=numthreads;
    for(i=0;i<numthreads;i++)
        pthread_create(&thrIds[i],NULL,callWrite,(void *)&strArgs);
    for(i=0;i<numthreads;i++)
        pthread_join(thrIds[i],NULL);

    controlDb(0);


    performanceStats(count,wrCost,1);
```

```
} else if (strcmp(argv[1], "read") == 0) {

    int r = 0;
    //assign given num of requests to count Var
    count = atoi(argv[2]);

    //prints
    _print_header(count);
    _print_environment();

    //argc=5 ->0..4
    if (argc == 5)
        r = 1;

    //synolika threads apo ton xristi
    numthreads = atoi(argv[3]);
    pthread_t thrIds[numthreads];
    controlDb(1);

    strArgs.rArg=r;
    strArgs.countArg=count;
    strArgs.numthr=numthreads;
    for(i=0;i<numthreads;i++)
        pthread_create(&thrIds[i],NULL,callRead,(void *)&strArgs);
    for(i=0;i<numthreads;i++)
        pthread_join(thrIds[i],NULL);

    controlDb(0);

    performanceStats(count,rdCost,2);
```

```
} else if (strcmp(argv[1], "addget") == 0) {
    int wrperc = 0;//pososto gia write
    int rdperc = 0;//pososto gia write


    //given num of threads for each operation
    int wrnumthreads = atoi(argv[4]);
    int rdnumthreads = atoi(argv[5]);

    int r = 0;
    //assign given num of requests to count Var
    count = atoi(argv[2]);

    //prints
    _print_header(count);
    _print_environment();


    //argc=7 -> 0..6
    if (argc == 7)
        r = 1;

    //synolika threads apo ton xristi
    numthreads = atoi(argv[3]);


    long int writeCount;
    long int readCount;


    controlDb(1);//anoigma mixanis

    wrperc= (wrnumthreads * 100 / numthreads);
    rdperc= (rdnumthreads * 100 / numthreads);



    //pososto count gia kathe leitoyrgia
    writeCount = (long int) (count*wrperc / 100); //synolika write poy ua ginoyn
    readCount = (long int) (count*rdperc / 100); //synolika read poy ua ginoyn


    createThreadsforAddGet(numthreads,wrnumthreads,rdnumthreads,writeCount,readCount,r,wrperc,rdperc);

    controlDb(0);

    performanceStats(writeCount,wrCost,1);
    performanceStats(readCount,rdCost,2);
```

(Δ)Ταυτοχρονισμός(ερώτημα 4)

Σε αυτό το στάδιο επεξεργαστήκαμε τα αρχεία db.c και db.h.

Αρχικά, στο αρχείο κεφαλίδας db.h δηλώσαμε κάποιες νέες μεταβλητές :

- την integer "diabastike"
- την boolean "write" που λειτουργεί ως flag
- 2 μεταβλητές συνθηκων τύπου pthread_cond_t "readOper" , "writeOper"
- Το mutex "krisimi"(επικοινωνεί και με τις 2 λειτουργίες add/get)

Έπειτα, επεξεργαστήκαμε το αρχείο db.c. Εδώ στόχος μας είναι η σωστή λειτουργία των db_add και db_get που καλούνται από τις συναρτήσεις _write_test και _read_test στο αρχείο kiwi.c

Στην συνάρτηση db_open_ex αρχικοποιούμε τις καινούργιες μεταβλητές που ορίσαμε στην db.h. Την int μεταβλητή σε 0, την bool σε false και τις υπόλοιπες δυναμικά χρησιμοποιώντας τις κατάλληλες ρουτίνες αρχικοποίησης.

Στην db_add, που είναι η συνάρτηση που υλοποιεί την λειτουργία write, στόχος μας είναι να υλοποιείται μόνο από έναν γραφέα ,όχι παράλληλα.

Δηλωσάμε μια τοπική μεταβλητή integer memadd, η οποία μας επιτρέπει να επιστρέψουμε το αποτέλεσμα της memtable_add μετά το πέρας της κρίσιμης περιοχής.

Ακόμα υλοποίησαμε μια συνάρτηση που καλούμε στην αρχή της db_add ώστε να φαίνονται οι αλλαγές που κάναμε. Την ονομάσαμε blockRead και αρχικά κλειδώνουμε το mutex "krisimi" πριν μπούμε στη κρίσιμη περιοχή. Με μία while και τη χρήση της μεταβλήτη "diabastike" στην συνθήκη ελέγχου ,σε περίπτωση που αυτή είναι μεγαλύτερη του 0 (σημαίνει υλοποιείται reading) τότε με τη ρουτίνα pthread_cond_wait(&self->readOper,&self->krisimi) αναστέλλουμε το νήμα και έτσι ¨μπλοκάρουμε¨ τους αναγνώστες ώστε να γράψουμε στην μηχανή μας. Μόλις βγούμε από τή while θέτουμε το flag write= true επειδή ακολουθεί η γραφή. Έπειτα επιστρέφουμε στην συνάρτηση db_add.

Γίνεται ο έλεγχος , ο οποίος προϋπήρχε, και σε αυτό το σημείο επιστρέφουμε στην μεταβλητή memadd το αποτέλεσμα της συνάρτησης memtable_add όπως είπαμε πριν. Τέλος υλοποιήσαμε μια ακόμα συνάρτηση την οποία καλούμε στο τέλος της db_add με όνομα giveSignal. Σε αυτή την συνάρτηση ξαναδηλώνουμε τη μεταβλητή write ίση με false εφόσον τελείωσε η γραφή.

Χρησιμοποιήσαμε τη ρουτίνα pthread_cond_signal(&self->writeOper) για την σηματοδότηση άλλου νήματος το οποίο περιμένει την μεταβλητή συνθήκης writeOper(ξεκλειδώνει το add) .Τέλος ξεκλειδώνουμε την mutex "krisimi" αφού η κρίσιμη περιοχή του κώδικα μας εκτελέστηκε και επιστρέφουμε τη τοπική μεταβλητή memadd με ένα return.

```
void blockWrite(DB* self){
    pthread_mutex_lock(&self->krisimi);
    while(self->write == true){
        pthread_cond_wait(&self->writeOper,&self->krisimi);
    }
    self->diabastike=self->diabastike+1;
    pthread_mutex_unlock(&self->krisimi);
}
```

```
void giveSignal(DB* self){
    self->write = false;
    pthread_cond_signal(&self->writeOper);
    pthread_mutex_unlock(&self->krisimi);
}
```

```
int db_add(DB* self, Variant* key, Variant* value)
{
    int memadd;
    //Lock: lh krisimi perioxi
    blockRead(self);
    if (memtable_needs_compaction(self->memtable))
    {
        INFO("Starting compaction of the memtable after %d insertions and %d deletions",
             self->memtable->add_count, self->memtable->del_count);
        sst_merge(self->sst, self->memtable);
        memtable_reset(self->memtable);
    }

    memadd = memtable_add(self->memtable, key, value);//boithitiki metabliti gia na ginei to unlock
    //self->writeFound += 1;
    //UnLock: lh krisimi perioxi
    giveSignal(self);
    return memadd;


}
```

Στην db_get, που είναι η συνάρτηση που υλοποιεί την λειτουργία read, στόχος μας είναι να διαβάζουνε πολλοί αναγνώστες παράλληλα.

Δηλώσαμε μια τοπική μεταβλητή integer met, η οποία μας επιτρέπει να επιστρέψουμε το αποτέλεσμα μετά το πέρας της κρίσιμης περιοχής.

Ακόμα υλοποιήσαμε μια συνάρτηση που καλούμε στην αρχή της db_get ώστε να φαίνονται οι αλλαγές που κάναμε και πάλι. Την ονομάσαμε blockWrite και αρχικά κλειδώνουμε το mutex "krisimi" πριν μπούμε στη κρίσιμη περιοχή. Με μία while και τη χρήση της μεταβλητή "write" στην συνθήκη ελέγχου ,σε

περίπτωση που αυτή ίση με true (σημαίνει υλοποιείται write) τότε με τη ρουτίνα pthread_cond_wait (&self->writeOper,&self->krisimi) αναστέλλουμε το νήμα και έτσι ¨μπλοκάρουμε¨ τους γραφείς ώστε να διαβάσουμε απ' την μηχανή μας. Μόλις βγούμε από τη while αυξάνουμε το diabastike κατά 1 για να καταγράψουμε τον επιπλέον γραφέα και ξεκλειδώνουμε τη κρίσιμη περιοχή επειδή ακολουθεί η ανάγνωση. Έπειτα επιστρέφουμε στην συνάρτηση db_get.

Στον ήδη υπάρχων έλεγχο δεν βγαίνουμε κατευθείαν απ'τή συνάρτηση αν ισχύει η συνθήκη if (return 1) αλλά επιστρέφουμε την τιμή της στην τοπική μεταβλητή που ορίσαμε αρχικά την met ώστε να την επιστρέψουμε μετά το τέλος της db_get.

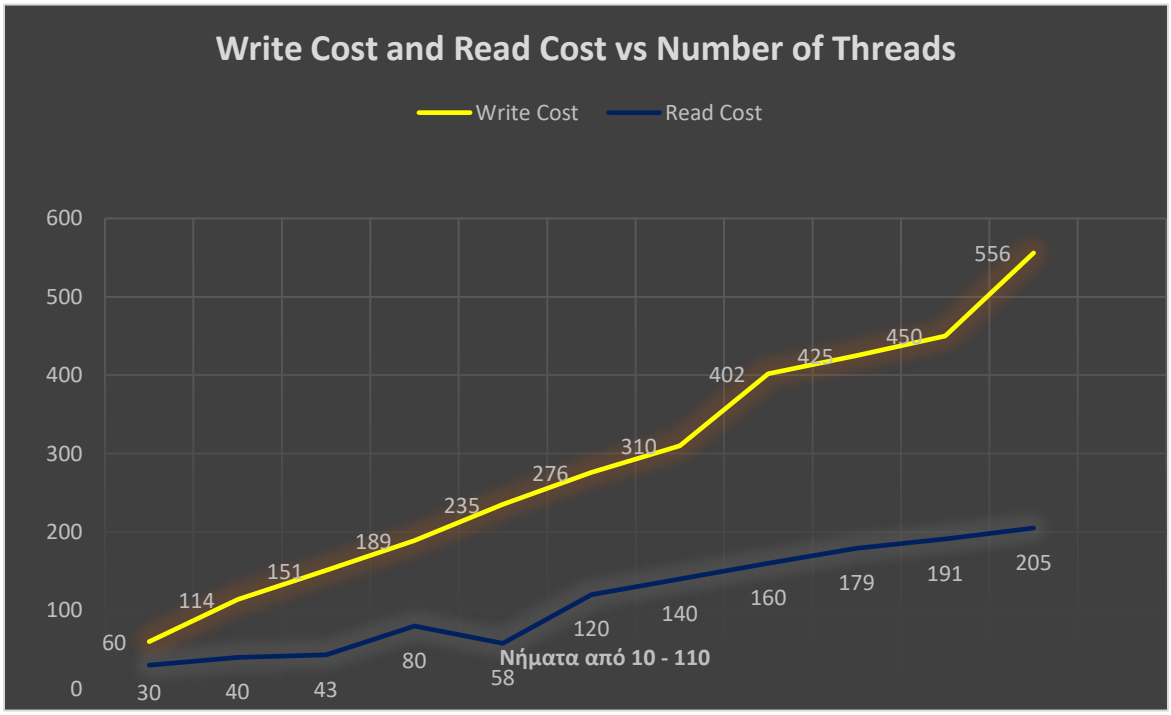
Τέλος υλοποιήσαμε μια ακόμα συνάρτηση την οποία καλούμε στο τέλος της db_get με όνομα giveSignal2.Σε αυτή την συνάρτηση μειώνουμε τη μεταβλητή diabastike κατά 1 με μια while όταν αυτή είναι ίση με το 0 χρησιμοποιήσαμε τη ρουτίνα pthread_cond_signal(&self->readOper) για την σηματοδότηση άλλου νήματος το οποίο περιμένει την μεταβλητή συνθήκης readOper(ξεκλειδώνει το get) .Τέλος ξεκλειδώνουμε την mutex "krisimi" αφού η κρίσιμη περιοχή του κώδικα μας εκτελέστηκε και επιστρέφουμε τη τοπική μεταβλητή met με ένα return.

(3)ΤΡΙΤΟ ΣΤΑΔΙΟ

(Ε) ΑΠΟΤΕΛΕΣΜΑΤΑ ΜΕΤΡΗΣΕΩΝ

Παρακάτω παρουσιάζουμε κάποιες αναπαραστάσεις του κόστους κάθε λειτουργίας συναρτήσει του αριθμού νημάτων. Ξεκινάμε με 10 νήματα αυξάνοντας κατά 10 κάθε φορά που ξανατρέχουμε και το δείγμα μας φτάνει μέχρι 120. Ορίσαμε τον αριθμό count ίσο με 500.000 σε όλες τις μετρήσεις που καταγράψαμε στα διαγράμματα. Στον άξονα x αντιστοιχούν τα νήματα και στον άξονα y το κόστος.

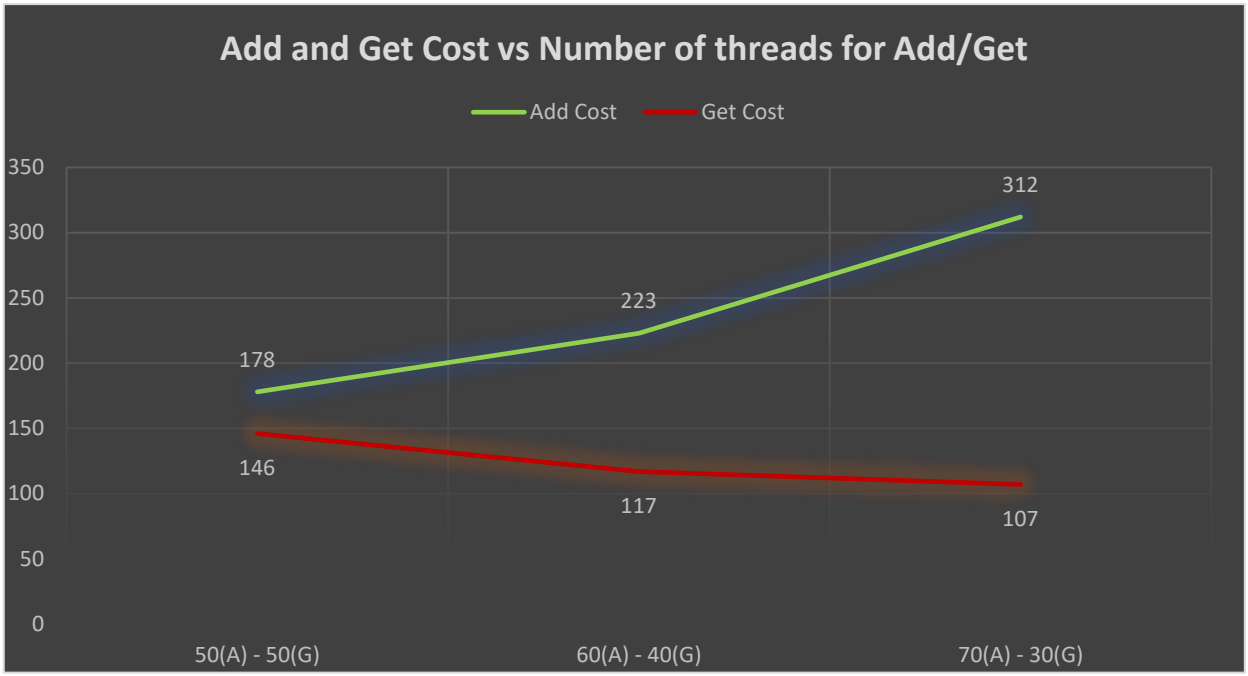
Για να είναι πιο ξεκάθαρο σε κάθε μέτρηση αναφέρονται οι ακριβείς μετρήσεις πάνω στις γραμμές.



Ειδικότερα στο 2ο διάγραμμα δώσαμε αριθμό νημάτων για 3 μετρήσεις .Στη πρώτη μέτρηση δώσαμε συνολικό αριθμό νημάτων 100 ,εκ των οποίων τα 50 υπεύθυνα για add(write) και τα άλλα 50 υπεύθυνα για get(read).

Στη δεύτερη μέτρηση δώσαμε συνολικό αριθμό νημάτων 100 ,εκ των οποίων τα 60 υπεύθυνα για add(write) και τα άλλα 40 υπεύθυνα για get(read).

Στη τρίτη μέτρηση δώσαμε συνολικό αριθμό νημάτων 100 ,εκ των οποίων τα 70 υπεύθυνα για add(write) και τα άλλα 30 υπεύθυνα για get(read).



ΕΚΤΕΛΕΣΕΙΣ ΣΤΟΝ ΤΕΡΜΑΤΙΚΟ

- Operation:write Count:400.000 Threads:30

PC1:

```
+----------------------------+---------------+----------------------------+-----------------
-+
-----------------------------------------------------------------------------------------------
--
|Random-WRITE    (done:400000): 0.000290 sec/op; 3448.3 writes/sec(estimated); cost:116.000(sec);
```

PC2:

```
+----------------------------+---------------+----------------------------+------------------+
-----------------------------------------------------------------------------------------------
|Random-WRITE    (done:400000): 0.000285 sec/op; 3508.8 writes/sec(estimated); cost:114.000(sec);
```

- Operation:read Count:400.000 Threads:30

PC1:

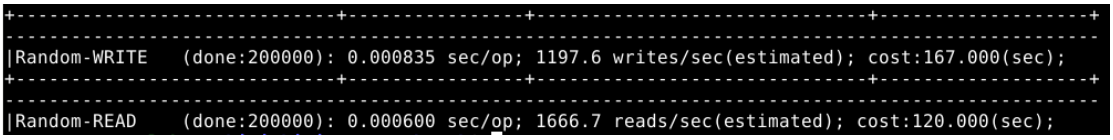
```
+----------------------------+---------------+----------------------------+-----------------
-+
-----------------------------------------------------------------------------------------------
--
|Random-READ    (done:400000): 0.000145 sec/op; 6896.6 reads/sec(estimated); cost:58.000(sec);
```

PC2:

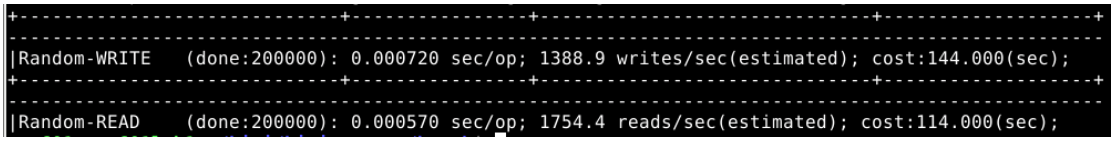
```
+----------------------------+---------------+----------------------------+------------------+
-----------------------------------------------------------------------------------------------
|Random-READ    (done:1000000): 0.000090 sec/op; 11111.1 reads/sec(estimated); cost:90.000(sec);
```

- Operation:addget Count:400.000 Threads:100 Add:50 Get:50

PC1 :

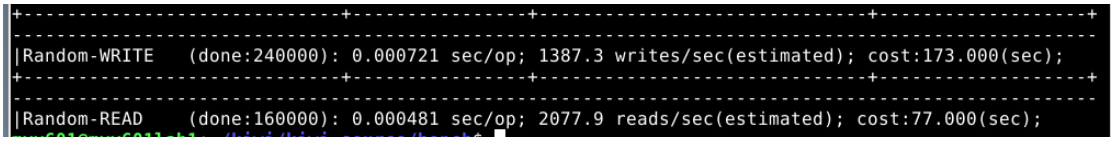


PC2:

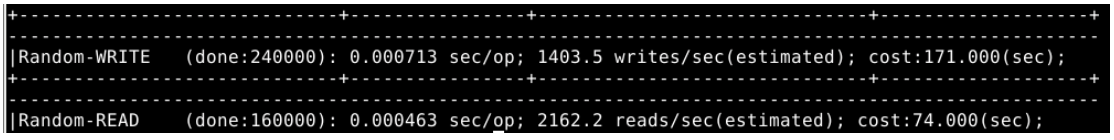


- Operation:addget Count:400.000 Threads:100 Add:60 Get:40

PC1:

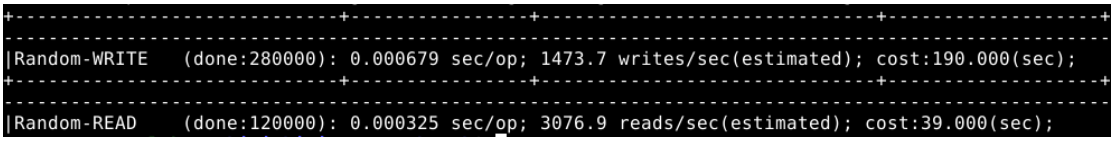


PC2:

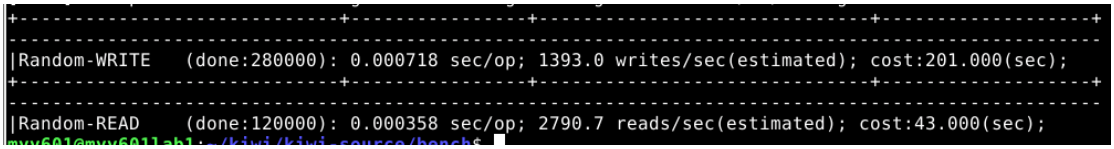


- Operation:addget Count:400.000 Threads:100 Add:70 Get:30

PC1:



PC2:



ΕΞΟΔΟΣ ΤΕΛΙΚΗΣ ΕΝΤΟΛΗΣ MAKE

```
myy601@myy601lab1:~/kiwi/kiwi-source$ make clean
cd engine && make clean
make[1]: Entering directory '/home/myy601/kiwi/kiwi-source/engine'
rm -rf *.o libindexer.a
make[1]: Leaving directory '/home/myy601/kiwi/kiwi-source/engine'
cd bench && make clean
make[1]: Entering directory '/home/myy601/kiwi/kiwi-source/bench'
rm -f kiwi-bench
rm -rf testdb
make[1]: Leaving directory '/home/myy601/kiwi/kiwi-source/bench'
myy601@myy601lab1:~/kiwi/kiwi-source$ make all
cd engine && make all
make[1]: Entering directory '/home/myy601/kiwi/kiwi-source/engine'
    CC db.o
    CC memtable.o
    CC indexer.o
    CC sst.o
    CC sst_builder.o
    CC sst_loader.o
    CC sst_block_builder.o
    CC hash.o
    CC bloom_builder.o
    CC merger.o
    CC compaction.o
    CC skiplist.o
    CC buffer.o
    CC arena.o
    CC utils.o
    CC crc32.o
    CC file.o
    CC heap.o
    CC vector.o
    CC log.o
    CC lru.o
    AR libindexer.a
make[1]: Leaving directory '/home/myy601/kiwi/kiwi-source/engine'
cd bench && make all
make[1]: Entering directory '/home/myy601/kiwi/kiwi-source/bench'
gcc -g -ggdb -Wall -Wno-implicit-function-declaration -Wno-unused-but-set-variable bench.c kiwi.c -L ../engine -lindexer -lpthread -lsnappy -o kiwi-bench
make[1]: Leaving directory '/home/myy601/kiwi/kiwi-source/bench'
myy601@myy601lab1:~/kiwi/kiwi-source$
```

(ΣΤ) ΑΣΤΟΧΙΕΣ

Κάθε φορά που τρέχουμε τις 3 διαφορετικές εντολές με την ακόλουθη σειρά read/write/addget κάνουμε ξανά make clean,make all στον φάκελο kiwi-source.Παρατηρήσαμε ότι αν τρέξουμε την ίδια εντολή με τα ίδια ορίσματα κάθε φορά υπάρχουν αποκλίσεις στους χρόνους και στα κόστη ,κατά το πλείστων προσεγγιστικά σε μία ακέραια τιμή .Μόνο ελάχιστες φορές το read ίσως χρειαστεί αρκετά λιγότερο χρόνο να διαβάσει αλλά σε περίπτωση που το ξανατρέξουμε τότε και αυτό πάλι προσεγγίζει την ακέραια τιμή. Αυτό το παρατηρήσαμε διότι τρέχαμε τις ίδιες εντολές η καθεμία στον δικό μας υπολογιστή για να δούμε πως συμπεριφέρεται η μηχανή. Επίσης παρατηρήσαμε ότι για πολύ μικρό αριθμό νημάτων ή count (requests) πραγματοποιούνται τα get(υποθέτουμε) τόσο γρηγορά ώστε να φαίνονται τα διαβάσματα άπειρα σε αριθμό ανά δευτερόλεπτο και συνεπώς και το κόστος ίσο με το 0. Ακόμα παρατηρήσαμε ότι στην addget όταν δίναμε τις μετρήσεις όπως στο διάγραμμα παραπάνω ,δηλαδή count=500.000 ,νήματα= 100 εκ των οποίων από 70 και πάνω για add και από 30 και κάτω για get τότε το πρόγραμμα σταματούσε με segmentation fault αλλά όταν το ξανατρέχαμε μια ή όσες φορές χρειαζόταν έβγαζε αποτέλεσμα κανονικά .Γενικότερα δεν

αντιμετωπίσαμε κάποιο άλλο πρόβλημα όσο πειραματιστήκαμε. Από τον κώδικα μέχρι και το report συνεργαστήκαμε παράλληλα και οι 2 ώστε να ακολουθούμε την ίδια γραμμή και να έχουμε πλήρη κατανόηση του πως λειτουργεί η μηχανή αποθήκευσης δεδομένων.