ΠΑΡΑΛΛΗΛΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΣΕ ΠΡΟΒΛΗΜΑΤΑ ΜΗΧΑΝΙΚΗΣ ΜΑΘΗΣΗΣ

3η Εργαστηριακή Άσκηση

Το πρόβλημα του περιοδεύοντα πωλητή, τυχαία αναζήτηση και ο αλγόριθμος ant colony

Ον/μο: Γεώργιος Σκόνδρας

Τμήμα: Μηχανικών Η/Υ και Πληροφορικής

Α.Μ: 1020408

Σημείωση

Για την υλοποίηση της άσκησης χρησιμοποιήθηκε προσωπικός φορητός υπολογιστής Huawei Matebook d14 με τα παρακάτω χαρακτηριστικά

- *Επεξεργαστής: AMD Ryzen 3500U*
- *Λειτουργικό Σύστημα: (μέσω Oracle VM Virtual Box 6.1) Ubuntu 20.04.1 LTS*
- *Compiler: gcc 9.3.0 (Ubuntu 9.3.0-17ubuntu1~20.04)*

Εφόσον χρησιμοποιήθηκε Virtual Machine για την εκπόνηση της άσκησης, δώσαμε 2 φυσικούς πυρήνες που αντιστοιχούν σε 4 thread(Simultaneous Multi Threading)

*Οι κώδικες για την κάθε εργασία βρίσκονται στα αρχεία *.c στους φακέλους ex1,ex2 κλπ*

Εργασία 1

Οι απαραίτητοι πίνακες που θα χρειαστούμε ώστε να αποθηκεύσουμε την πληροφορία που χρειαζόμαστε είναι οι εξής:

- **int Cities[10000][2]** : Δισδιάστατος πίνακας που περιέχει που σε κάθε γραμμή περιέχει τις συντεταγμένες της κάθε πόλης. Έχει 10000 γραμμές, όσες κι οι πόλεις και 2 στήλες, καθώς οι θέσεις των πόλεων είναι στις 2 διαστάσεις.
- **int Route[10000]** : Μονοδιάστατος πίνακας που περιέχει την τρέχουσα διαδρομή. Έχει 10000 γραμμές, όσες κι οι πόλεις. πχ στη θέση Route[0] βρίσκεται ο αριθμός της πόλης στην αρχή της διαδρομή, στη θέση Route[1] ο αριθμός της αμέσως επόμενης πόλης κοκ.

Οι απαραίτητες συναρτήσεις που θα χρειαστούμε είναι οι εξής:

- **void createCities()** : Συνάρτηση που αρχικοποιεί τις τιμές των πόλεων, δηλαδή του πίνακα Cities με τη χρήση της συνάρτησης rand().
- **void createRoute()** : Συνάρτηση που δημιουργεί μια τυχαία αρχικοποίηση του μονοπατιού. Επιλέγεται η αρχική σειρά των πόλεων, η οποία είναι τυχαία, καθώς και οι πόλεις παράγονται τυχαία.
- **void calculateDistance()** : Συνάρτηση η οποία υπολογίζει την συνολική ευκλείδεια απόσταση της τρέχουσας διαδρομής σύμφωνα με τον πίνακα Route
- **void swapCities(int city1, int city2)** : Συνάρτηση η οποία εναλλάσσει 2 πόλεις στη διαδρομή.

Θα κάνουμε επιπλέον χρήση 2 βοηθητικών συναρτήσεων void **printCities()** και void **printRoutes()** για να τυπώνουμε τις τιμές των πόλεων και του μονοπατιού.

Για να επιβεβαιώσουμε τη ορθή λειτουργία, τρέχουμε τον κώδικα για λίγες πόλεις και λίγες επαναλήψεις και παρατηρούμε πως ο αλγόριθμος εκτελείται σωστά, όπως φαίνεται παρακάτω.

Ο κώδικας βρίσκεται στο αρχείο tsp1.c και λειτουργεί ως εξής:

Εκτελούμε ένα for loop για **NUM_ITERATIONS** επαναλήψεις τις οποίες κάνουμε define στην αρχή του αρχείου δίνοντας την τιμή 20000. Στη συνέχεια επιλέγουμε τυχαία 2 πόλεις και τις κάνουμε εναλλαγή στη διαδρομή. Αν η νέα απόσταση που προκύπτει στη διαδρομή είναι μεγαλύτερη από αυτήν που υπήρχε πριν, τότε γίνεται πάλι εναλλαγή των πόλεων ώστε να μην αλλάξει η διαδρομή και προχωράμε στην επόμενη επανάληψη.

Για να υπολογίσουμε ποια τμήματα του προγράμματος καταναλώνουν το μεγαλύτερο χρόνο υπολογισμού θα κάνουμε profiling το σειριακό κώδικα όπως φαίνεται παρακάτω.

```
osboxes@osboxes:~/Desktop/parProg/lab3/ex1$ gcc tsp1.c -o tsp1 -lm -pg
osboxes@osboxes:~/Desktop/parProg/lab3/ex1$ time ./tsp1
Number of cities: 10000
Grid: 1000X1000

Initial Distance: 5231436.358306
Number of iterations: 20000
Number of succesful swaps:5320
Number of failing swaps:14680
Final Distance: 3493734.923092

real    0m8,719s
user    0m8,710s
sys     0m0,007s
osboxes@osboxes:~/Desktop/parProg/lab3/ex1$ gprof tsp1 gmon.out > analysis.txt
osboxes@osboxes:~/Desktop/parProg/lab3/ex1$ cat analysis.txt
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
100.55      3.26     3.26    40002    81.44    81.44  calculateDistance
  0.00      3.26     0.00    34680     0.00     0.00  swapCities
  0.00      3.26     0.00        1     0.00     0.00  createCities
  0.00      3.26     0.00        1     0.00     0.00  createRoute
```

Σύμφωνα με τον **profiler** το 100% το χρόνου δαπανήθηκε εκτελώντας τη συνάρτηση calulateDistance() η οποία υπολογίζει τις αποστάσεις πριν και μετά τις εναλλαγές των πόλεων. Το γεγονός αυτό είναι απόλυτα λογικό, καθώς οι άλλες συναρτήσεις είτε αρχικοποιούν τα δεδομένα και τρέχουν μόνο μία φορά ή κάνουν πολύ λίγες πράξεις εναλλαγής τιμών σε πίνακες.

Στην παραπάνω εικόνα παρατηρούμε ότι για 20000 επαναλήψεις του αλγορίθμου για τις 10000 πόλεις η τελική απόσταση είναι **distance=3493734km.** Να σημειωθεί πως η αρχική απόσταση ήταν **initial_distance=5231426km**.

Ο **χρόνος εκτέλεσης** είναι **8.7sec**. Το αποτέλεσμα δεν είναι αρκετά καλό, καθώς σε 8.7sec δεν καταφέραμε να πετύχουμε ούτε τη μισή απόσταση διαδρομής σε σχέση με την αρχική τυχαία διαδρομή. Αυτό είναι λογικό, καθώς κάνουμε τυχαία αναζήτηση και όπως φαίνεται παραπάνω από τις 20000 εναλλαγές πόλεων μόνο οι 5000 περίπου πέτυχαν να μειώσουν την απόσταση της διαδρομής.

Η **μνήμη** που δεσμεύεται φαίνεται παρακάτω:

- Πίνακες:
    - int Cities[10000][2]
    - int Route[10000]
- Μεταβλητές :
    - int i, city1, city2(βοηθητικές οι int success, fail)
    - double oldDistance, newDistance

Εργασία 2

Στην εργασία αυτή θα παραλληλοποιήσουμε τον σειριακό αλγόριθμο του προηγούμενου ερωτήματος. Εφόσον το 100% του χρόνου καταναλώνεται στη συνάρτηση calculateDistance() θα παραλληλοποιήσουμε αυτήν και πιο συγκεκριμένα το διπλό for loop που υπολογίζονται οι αποστάσεις. Θα προσθέσουμε την παρακάτω οδηγία της OMP:

```
#pragma omp parallel for private(j,tempDistance) reduction(+:distance)
```

- private(j,tempdistance) , διότι οι δύο αυτές μεταβλητές χρησιμοποιούνται στο εμφωλευμένο for loop
- reduction(+:distance) , διότι distance είναι η συνολική αθροιστική απόσταση της διαδρομής και το κάθε thread πρέπει να προσθέσει σε αυτήν το επιμέρους άθροισμα.

Τρέχουμε το πρόγραμμα για αριθμό επαναλήψεων NUM_ITERATIONS = 20000 όπως και πριν και παρατηρούμε τα αποτελέσματα.

```
osboxes@osboxes:~/Desktop/parProg/lab3/ex2$ gcc tsp2.c -o tsp2 -lm -fopenmp
osboxes@osboxes:~/Desktop/parProg/lab3/ex2$ time ./tsp2
Number of cities: 10000
Grid: 1000X1000

Number of available threads:4
Initial Distance: 5231436.358306
Number of iterations: 20000
Number of succesful swaps:5320
Number of failing swaps:14680
Final Distance: 3493734.923092

real    0m3,388s
user    0m13,384s
sys     0m0,041s
osboxes@osboxes:~/Desktop/parProg/lab3/ex2$
```

Παρατηρούμε ακριβώς τα ίδια αποτελέσματα με το σειριακό πρόγραμμα που μας διαβεβαιώνει για την ορθή εκτέλεση του παράλληλου προγράμματος. Βλέπουμε ότι ο χρόνος απόκρισης βελτιώθηκε σημαντικά.

Πλέον η εκτέλεση έγινε μόλις σε **χρόνο** περίπου **3.4s**. Σχεδόν υποτριπλασιάσαμε το χρόνο εκτέλεσης με χρήση 4 thread. Η μνήμη που χρησιμοποιήθηκε είναι η ίδια με το προηγούμενο ερώτημα. Οι επιπλέον private μεταβλητές για κάθε thread έχουν ανεπαίσθητη διαφορά στη συνολική δεσμευμένη μνήμη.

Παρατηρούμε ότι η συνολική απόσταση **3493734km** είναι ίδια με τη σειριακή.

Βέβαια εφόσον η εκτέλεση είναι γρηγορότερη, μπορούμε να αυξήσουμε τον αριθμό των επαναλήψεων. Έτσι για 50000 επαναλήψεις σε σχέση με τις 20000 παρατηρούμε τα παρακάτω:

```
osboxes@osboxes:~/Desktop/parProg/lab3/ex2$ time ./tsp2
Number of cities: 10000
Grid: 1000X1000

Number of available threads:4
Initial Distance: 5231436.358306
Number of iterations: 50000
Number of succesful swaps:8784
Number of failing swaps:41216
Final Distance: 2848385.882037

real    0m8,859s
user    0m33,873s
sys     0m0,091s
```

Βλέπουμε ότι σε **8.8sec** όσο περίπου έκανε η σειριακή εκτέλεση πετύχαμε μικρότερη συνολική απόσταση **distance=2848385**

Εργασία 3

Οι πίνακες που θα χρειαστούμε θα είναι και αυτοί που χρησιμοποιήσαμε στα προηγούμενα ερωτήματα, δηλαδή:

- **int Cities[10000][2]**
- **int Route[10000]**

Επιπλέον θα κάνουμε χρήση ενός πίνακα

- **int cityTaken[10000]** : Μονοδιάστατος πίνακας-μάσκα που περιέχει την πληροφορία αν μία πόλη έχει ήδη επιλεγεί ή όχι στη διαδρομή. Αν μία πόλη έχει επιλεγεί ήδη στη διαδρομή, η αντίστοιχη τιμή του πίνακα αυτό είναι 1, αλλιώς είναι 0.

Οι βασικές συναρτήσεις που εκτελούν τους απαραίτητους υπολογισμούς είναι οι εξής:

- **double twoCitiesDistance(int city1, int city2)** : Συνάρτηση που παίρνει σαν παραμέτρους 2 πόλεις και επιστρέφει την μεταξύ τους ευκλείδεια απόσταση.
- **int closerCity(int city)** : Συνάρτηση που παίρνει σαν παράμετρο μία πόλη και επιστρέφει την πιο κοντινή σε αυτήν πόλη.

Επιβεβαιώνουμε την ορθή λειτουργία του αλγορίθμου με μικρό αριθμό πόλεων, όπως φαίνεται παρακάτω:

```
osboxes@osboxes:~/Desktop/parProg/lab3/ex3$ gcc tsp3.c -o tsp3 -lm -Wall
osboxes@osboxes:~/Desktop/parProg/lab3/ex3$ ./tsp3
Number of cities: 5
Grid: 5X5

Cities:

0: 3 1
1: 2 0
2: 3 0
3: 1 2
4: 4 1

Route:

0 -> 0 -> 0 -> 0 -> 0 -> 0

Route:

0 -> 2 -> 0 -> 0 -> 0 -> 0

Route:

0 -> 2 -> 1 -> 0 -> 0 -> 0

Route:

0 -> 2 -> 1 -> 3 -> 0 -> 0

Route:

0 -> 2 -> 1 -> 3 -> 4 -> 0

Final Distance:8.398346
osboxes@osboxes:~/Desktop/parProg/lab3/ex3$
```

Τρέχουμε τον κώδικα για 10000 πόλεις και παρατηρούμε τον χρόνο απόκρισης και τη συνολική απόσταση.

```
osboxes@osboxes:~/Desktop/parProg/lab3/ex3$ gcc tsp3.c -o tsp3 -lm -Wall
osboxes@osboxes:~/Desktop/parProg/lab3/ex3$ time ./tsp3
Number of cities: 10000
Grid: 1000X1000

Final Distance:88438.428396

real    0m1,158s
user    0m1,152s
sys     0m0,007s
osboxes@osboxes:~/Desktop/parProg/lab3/ex3$
```

Όσον αφορά τον χρόνο εκτέλεσης, από τα 3,4sec της Εργασίας 2, με αυτή την υλοποίηση έχουμε **χρόνο εκτέλεσης 1.1sec**. Η βελτίωση είναι σημαντική, καθώς ο αλγόριθμος τρέχει 3 φορές πιο γρήγορα. Όσον αφορά την απόσταση, στην Εργασία 2 είχαμε πετύχει απόσταση **distance=3493734** για 20000 επαναλήψεις και **distance=2848385**. Σε αυτή την υλοποίηση πετύχαμε εξαιρετική μείωση της απόστασης, καθώς πλέον έχουμε απόσταση **distance=88438**. Η απόσταση μειώθηκε περίπου 30 φορές!

Εργασία 4

Για να υλοποιήσουμε τον αλγόριθμο που ζητείται σε αυτό το ερώτημα δε θα χρειαστούμε κάποια νέα συνάρτηση. Η αλλαγή που θα κάνουμε είναι ότι σε κάθε βήμα βρίσκουμε τις δύο κοντινότερες πόλεις και διαλέγουμε τυχαία μία από αυτές. Η αλλαγή έχει γίνει στη main. Η πιθανότητες p1,p2 που επιλέγουμε τις εκάστοτε πόλεις φαίνονται παρακάτω.

$$p1 = \frac{a}{a+1}\,, p2 = \frac{1}{a+1}\,, με\ a > 1\ ώστε\ p1 > p2$$

P1 είναι η πιθανότητα να επιλεγεί η κοντινότερη πόλη και P2 να επιλεγεί η αμέσως κοντινότερη πόλη. Δίνουμε στην παράμετρο α την τιμή $\alpha = 10$ που οδηγεί σε πιθανότητες

$p1 \cong 0.91\ και\ p2 \cong 0.09$ και επιβεβαιώνουμε την σωστή λειτουργία εκτελώντας το πρόγραμμα για μικρό αριθμό πόλεων, όπως φαίνεται παρακάτω.

```
osboxes@osboxes:~/Desktop/parProg/Labs/lab3/ex4$ gcc tsp4.c -o tsp4 -lm
osboxes@osboxes:~/Desktop/parProg/Labs/lab3/ex4$ ./tsp4
a = 10.000000
p1=0.909091 p2=0.090909
Cities:

0: 7 6
1: 1 3
2: 1 7
3: 2 4
4: 1 5

Route:

0 -> 3 -> 0 -> 0 -> 0 -> 0

Route:

0 -> 3 -> 1 -> 0 -> 0 -> 0

Route:

0 -> 3 -> 1 -> 4 -> 0 -> 0

Route:

0 -> 3 -> 1 -> 4 -> 2 -> 0

Final Distance:16.882141
```

Στη συνέχεια εκτελούμε για τον ίδιο αριθμό πόλεων με τα προηγούμενα ερωτήματα.

```
osboxes@osboxes:~/Desktop/parProg/Labs/lab3/ex4$ gcc tsp4.c -o tsp4 -lm
osboxes@osboxes:~/Desktop/parProg/Labs/lab3/ex4$ time ./tsp4
a = 10.000000
p1=0.909091 p2=0.090909
Number of cities: 10000
Grid: 1000X1000

Final Distance:94595.201155

real    0m2,915s
user    0m2,907s
sys     0m0,008s
osboxes@osboxes:~/Desktop/parProg/Labs/lab3/ex4$
```

Στη συνέχεια κάνουμε profiling για να δούμε ποια τμήματα του κώδικα καταναλώνουν το μεγαλύτερο χρόνο, όπως φαίνεται παρακάτω.

```
osboxes@osboxes:~/Desktop/parProg/Labs/lab3/ex4$ gcc tsp4.c -o tsp4 -lm -pg
osboxes@osboxes:~/Desktop/parProg/Labs/lab3/ex4$ time ./tsp4
a = 10.000000
p1=0.909091 p2=0.090909
Number of cities: 10000
Grid: 1000X1000

Final Distance:96062.704616

real    0m3,403s
user    0m3,396s
sys     0m0,007s
osboxes@osboxes:~/Desktop/parProg/Labs/lab3/ex4$ gprof tsp4 gmon.out > analysis.txt
osboxes@osboxes:~/Desktop/parProg/Labs/lab3/ex4$ cat analysis.txt
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 60.56      0.85     0.85 99980001     0.01     0.01  twoCitiesDistance
 39.90      1.42     0.56    19998    28.13    70.83  closerCity
  0.00      1.42     0.00        2     0.00     0.00  calculateDistance
  0.00      1.42     0.00        1     0.00     0.00  createCities
```

Σύμφωνα με τον profiler, τον περισσότερο χρόνο τον καταναλώνουν οι παρακάτω συναρτήσεις:

- twoCitiesDistance: 60.56% Η συνάρτηση που υπολογίζει την ευκλείδεια απόσταση μεταξύ δύο πόλεων
- closerCity: 39.9% Η συνάρτηση που επιστρέφει την κοντινότερη πόλη

Όσον αφορά το χρόνο εκτέλεσης, στην πρώτη εκτέλεση που δεν κάναμε profiling είχαμε **χρόνο 3.4sec**, ο οποίος είναι ίδιος με το χρόνο που έκανε το πρόγραμμα της εργασίας 2 για 20000 επαναλήψεις.

Βέβαια η συνολική **απόσταση** είναι αρκετά μικρότερη από αυτή της Εργασίας 2, καθώς από **distance=3493734** τώρα η απόσταση έχει μειωθεί σε **distance=94595**

Η μνήμη που χρησιμοποιήθηκε είναι ακριβώς η ίδια με αυτήν στην Εργασία 3

Εργασία 5

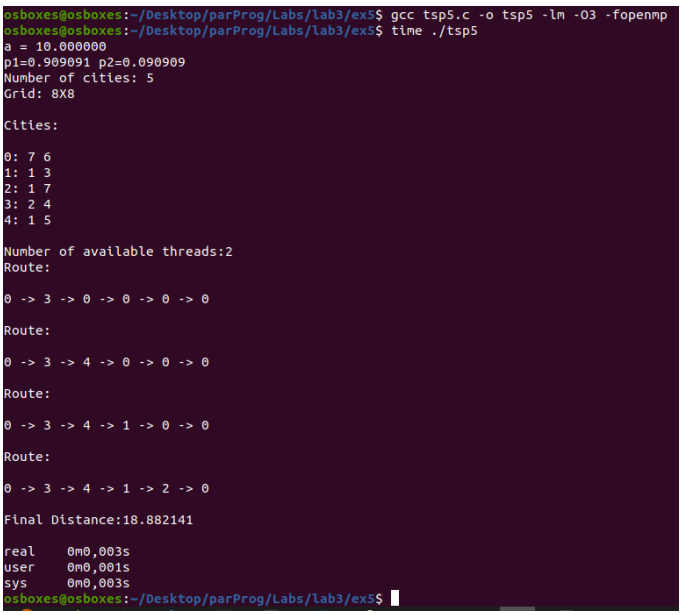
Στον κώδικα της Εργασίας 4 δεν μπορούμε να παραλληλοποιήσουμε το κύριο for loop στη main. Ο λόγος είναι ότι σε κάθε βήμα χρειαζόμαστε το αμέσως προηγούμενο, οπότε η εκτέλεση των εντολών εκεί πρέπει να εκτελεστούν σειριακά.

Η παραλληλοποίηση της συνάρτησης twoCitiesDistance, δεν είχε αποτέλεσμα στη μείωση του χρόνου εκτέλεσης. Αντίθετα ο χρόνος αυξήθηκε σημαντικά. Ο λόγος είναι ότι η συνάρτηση αυτή περιέχει μόνο ένα for loop το οποίο τρέχει μόνο για 2 iteration και η συνάρτηση σύμφωνα με τον profiler έχει περίπου 99 εκατομμύρια calls. Οπότε το overhead για create και join των thread είναι απαγορευτικό.

Συνεπώς θα παραλληλοποιηθεί μόνο η συνάρτηση closerCity με τα κατάλληλα OpenMP directives. Ο κώδικας βρίσκεται στο αρχείο tsp5.c

Επιπρόσθετα έγινε χρήση της εντολής srand(time(0)) μετά την αρχικοποίηση των πόλεων, ώστε να δίνουμε κάθε φορά διαφορετικό seed στη rand() και κάθε φορά να έχουμε διαφορετική εκτέλεση, αλλά πάνω στα ίδια δεδομένα.

Επιβεβαιώνουμε την σωστή λειτουργία εκτελώντας το για λίγες πόλεις όπως φαίνεται παρακάτω. Βλέπουμε πως έχουμε το ίδιο αποτέλεσμα με τη σειριακή εκτέλεση.



Στη συνέχεια εκτελούμε το πρόγραμμα για 10000 πόλεις και παρατηρούμε τα αποτελέσματα

Παρατηρούμε ακριβώς τα ίδια αποτελέσματα αποστάσεων με τον σειριακό κώδικα. Βέβαια ο χρόνος έχει αυξηθεί σε σχέση με το σειριακό. Αυτό οφείλεται στους ίδιους λόγους που είπαμε αμέσως παραπάνω ότι απέτυχε και η παραλληλοποίηση της twoCitiesDistance. Για την συγκεκριμένη υλοποίηση απαιτείται συγχρονισμός που καθυστερεί αρκετά την παράλληλη υλοποίηση ενώ η σειριακή εκτέλεση είναι ταχύτερη.

Εργασία 6

Σε αυτό το ερώτημα θα υλοποιήσουμε μόνοι μας το λογισμικό για την επίλυση του TSP με τη χρήση του Ant Colony Optimization. Πιο συγκεκριμένα θα υλοποιήσουμε μία απλοποιημένη έκδοση του αλγορίθμου που παρουσιάζεται στις εργασίες (Dorigo and Gambardella 1997; Yang et al. 2008).

Παρακάτω εξηγούμε τη λογική του αλγορίθμου που υλοποιήθηκε, τους πίνακες και τις βασικές συναρτήσεις που χρησιμοποιήθηκαν.

Οι πίνακες που χρησιμοποιήθηκαν είναι οι εξής:

- **int Cities[10000][2]**: Πίνακας με τις συντεταγμένες των πόλεων, ακριβώς ίδιος με τα προηγούμενα ερωτήματα.
- **int Route[NUM_ANTS][10000]:**Πίνακας που περιέχει τη διαδρομή κάθε μυρμηγκιού. Είναι στην ίδια λογική με τον πίνακα Route των προηγούμενων ερωτημάτων, απλά θα χρειαστούμε NUM_ANTS γραμμές, όπου NUM_ANTS ο αριθμός των μυρμηγκιών, για να αναπαραστήσουμε τη διαδρομή του κάθε μυρμηγκιού.
- **cityTaken[NUM_ANTS][NUM_CITIES]:** Πίνακας που περιέχει την πληροφορία αν ένα μυρμήγκι έχει επισκευθεί μια πόλη. Είναι στην ίδια λογική με τον πίνακα cityTaken των προηγούμενων ερωτημάτων, απλά θα χρειαστούμε NUM_ANTS γραμμές, όπου NUM_ANTS ο αριθμός των μυρμηγκιών, για να αναπαραστήσουμε την πληροφορία για κάθε μυρμήγκι.
- **double pherormone[10000][10000]:** Πίνακας που περιέχει την τιμή της φερορμόνης που περιέχει κάθε ακμή του γραφήματος των πόλεων. Εφόσον οι ακμές είναι διπλής κατεύθυνσης μας αρκεί μόνο το άνω τριγωνικό κομμάτι του πίνακα, χωρίς τη διαγώνιο.

Στην αρχή του αλγορίθμου αρχικοποιούμε τις πόλεις με τον ίδιο τρόπο που κάναμε και στα προηγούμενα ερωτήματα με τη χρήση της συνάρτησης **createCities()**.

Στη συνέχεια η συνάρτηση **initializePherormones()** αρχικοποιεί τις τιμές της φερορμόνης για κάθε ακμή του γραφήματος με την τιμή της σταθεράς **INITIAL_PHERORMONE**.

Στη συνέχεια με ένα for loop το οποίο τρέχει για **NUM_ITERATIONS** επαναλήψεις προσομοιώνουμε του γύρους που θα εκτελέσουμε τον αλγόριθμο.

Σε κάθε γύρο του αλγορίθμου πραγματοποιούμε τα εξής:

1. Τοποθετούμε τα «μυρμήγκια» μας σε τυχαίες πόλεις με τη χρήση της συνάρτησης **deployAnts()**

2. Πραγματοποιούμε ένα tour για όλα τα μυρμήγκια με τη συνάρτηση **allAntsTour()** στο οποίο το κάθε μυρμήγκι βρίσκει μία διαδρομή ανάμεσα σε όλες τις πόλεις.

Η συνάρτηση **allAntsTour()** προσομοιώνει την παραπάνω λειτουργικότητα με ένα for loop με NUM_ANTS επαναλήψεις, όσα και τα μυρμήγκια και σε κάθε επανάληψη καλεί τη συνάρτηση **singleAntTout()** με παράμετρο τον αριθμό του μυρμηγκιού. Μετά την ολοκλήρωση του loop, δηλαδή ενός γύρου, επαναφέρονται οι τιμές του πίνακα **cityTaken()** ώστε να μπορούμε να τρέξουμε ξανά τον επόμενο γύρο και γίνεται update το καλύτερο Route που έχει βρεθεί μέχρι στιγμής με τη χρήση της συνάρτησης **updateBestRoute().**

Η συνάρτηση **singleAntTour()** βρίκει μία διαδρομή για κάθε μυρμήγκι ανάμεσα σε όλες τις πόλεις. Σε κάθε βήμα η επόμενη πόλη υπολογίζεται με βάση μία ευρετική που εξαρτάται από την απόσταση μεταξύ της τωρινής πόλης από τις υπόλοιπες, καθώς και της τιμής της φερορμόνης στην ακμή που ενώνει τις 2 πόλεις. Η λογική αυτή υλοποιείται στη συνάρτηση **nextCity()** η οποία για κάθε πόλη υπολογίζει ένα score με τη συνάρτηση **cityScore()** και επιστρέφει την πόλη με το μεγαλύτερο score.

Το score της ακμής ανάμεσα στις πόλεις i,j δίνεται από τον τύπο:

$$score(i,j) = p(i,j) * v(i,j)\text{ , όπου}$$

$$p(i,j)\text{: Η τιμή της φερορμόνης στην ακμή } i \leftrightarrow j$$

$$v(i,j)\text{: Η τιμή του visibility στη από την πόλη } i \text{ στην } j\text{ , με}$$

$$v(i,j) = \frac{1}{distance(i,j)} \text{ και } distance(i,j) \text{ η ευκλείδια απόσταση των πόλεων } i,j$$

Σε κάθε μετακίνηση το κάθε μυρμήγκι στο βήμα k+1 από την πόλη i στην πόλη j ανανεώνει και την τιμή της φερορμόνης στην ακμή i,j σύμφωνα με την τιμής της φερορμόνης στο βήμα k, όπως φαίνεται στον παρακάτω τύπο:

$$p(i,j)_{k+1} = (1-a) * p(i,j)_k + a * v(i,j)\text{ , όπου}$$

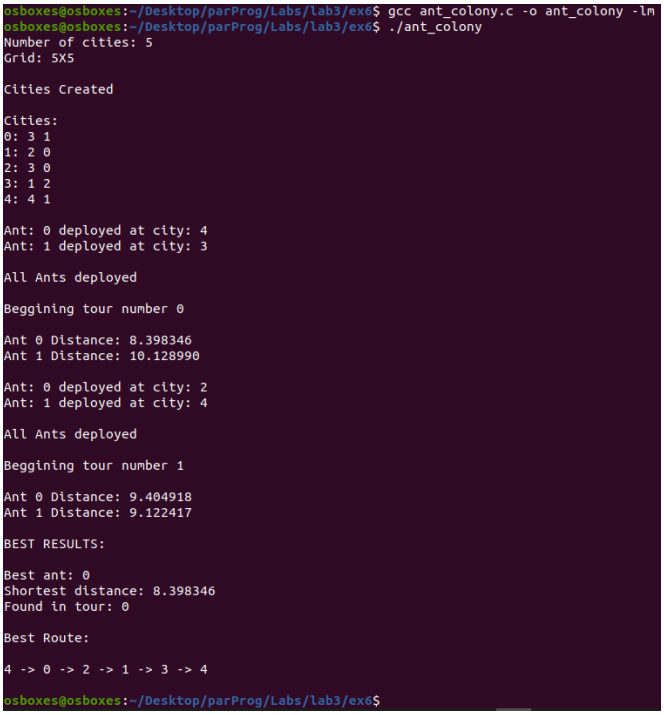
$$p(i,j)_k\text{: η τιμή της φερορμόνης στο βήμα } k$$

$$v(i,j)\text{: η τιμή του visibility από την πόλη } i \text{ στην } j\text{, όπως ορίστηκε παραπάνω}$$

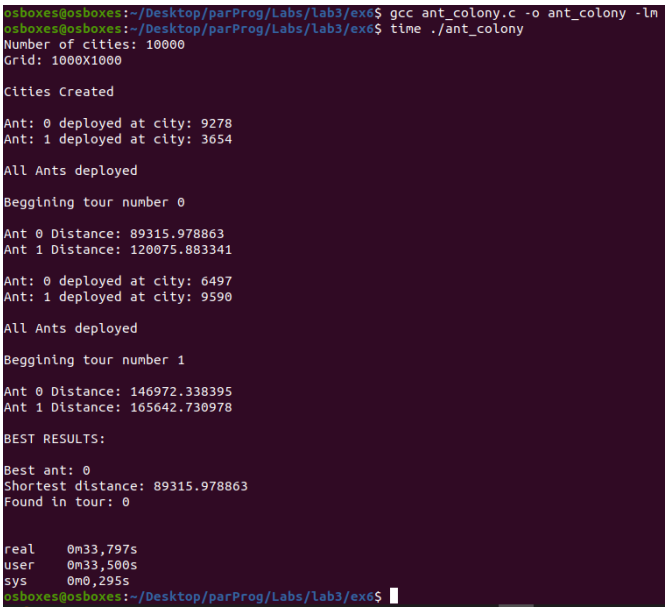
$$a\text{: παράμετρος, ορίστηκε σε } 0.9$$

Ο κώδικας βρίσκεται στα αρχείο ant_colony.c στο φάκελο ex6

Αρχικά επιβεβαιώνουμε τη σωστή λειτουργία του αλγορίθμου για μικρό αριθμό πόλεων, όπως και στα προηγούμενα ερωτήματα, όπως φαίνεται παρακάτω για 5 πόλεις, 2 μυρμήγκια και 2 γύρους.



Στη συνέχεια εκτελούμε τον αλγόριθμο για 10000 πόλεις. Αυτή τη φορά θα κάνουμε χρήση 2 μυρμηγκιών και 2 επαναλήψεων.



Ο **συνολικός χρόνος απόκρισης** σε σχέση με αυτόν της Εργασίας 2 έχει αυξηθεί σημαντικά, πιο συγκεκριμένα, για μόλις 2 μυρμήγκια και 2 γύρους απαιτείται **χρόνος 33sec** σε σχέση με τους χρόνους **3.4sec και 8.8sec** για 20000 και 50000 επαναλήψεις στον αλγόριθμο της Εργασίας 2.

Η **συνολική απόσταση** που έχει να διανύσει ο πωλητής κυμαίνεται ανάμεσα στα **89315 και 165642km** σε σχέση με τα **3493734km** και **2848385km** της Εργασίας 2.

Το μέγεθος της μνήμης που χρησιμοποιείται είναι σημαντικά μεγαλύτερο από αυτό της Εργασίας 2, αλλά και σε σχέση με οποιαδήποτε προηγούμενη Εργασία. Αυτό οφείλεται πως απαιτούνται μεγάλοι πίνακες για να αποθηκεύσουμε τις πληροφορίες των φερορμονών, καθώς και των διαδρομών του κάθε μυρμηγκιου.

Μόνο ο πίνακας cities παραμένει ίδιος ανάμεσα στις Εργασίες 2 και 6.

Ο πίνακας int Route[10000] πλέον έχει γίνει int Route[NUM_ANTS][10000], ώστε να αποθηκεύουμε τις διαδρομές κάθε μυρμηγκιού.

Επιπλέον χρησιμοποιήθηκαν οι παρακάτω πίνακες που αναλύθηκαν παραπάνω.

- cityTaken[NUM_ANTS][NUM_CITIES]
- double pherormone[10000][10000]

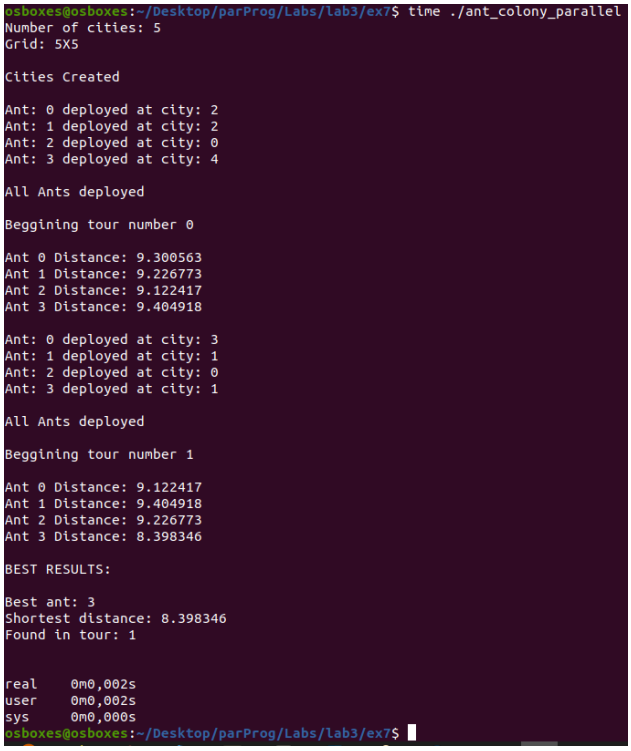
Εργασία 7

Η υλοποίηση του σειριακού κώδικα μας επιτρέπει αρκετά εύκολα να τον παραλληλοποιήσουμε. Πιο συγκεκριμένα η υλοποίηση έγινε με σκοπό το κάθε μυρμήγκι να μην κάνει προσπέλαση σε περιοχές μνήμης κοινές με άλλα μυρμήγκια. Κάνει μόνο προσπελάσεις στις γραμμές των πινάκων που σχετίζονται με το ίδιο το μυρμήγκι. Μοναδική εξαίρεση η προσπέλαση των τιμών του πίνακα των φερορμονών που εκεί έγινε χρήση της οδηγίας #pragma omp critical για να αποφύγουμε τα race conditions. Η δομή αυτή του κώδικα μας επιτρέπει να δούμε σημαντική επιτάχυνση παραλληλοποιώντας το for loop της συνάρτησης allAntsTour() μέσα στο οποίο κάνει tour το κάθε μυρμήγκι. Πρακτικά για τα 4 threads που διαθέτει το μηχάνημά μου, αν εκτελέσω τον αλγόριθμο για 4 μυρμήγκια, το κάθε μυρμήγκι τρέχει σε δικό του thread.

Για να γίνει αυτό, θα χρειαστεί πρώτα να κάνουμε κάποιες αλλαγές. Ο καθαρισμός του πίνακα που περιέχει την πληροφορία για τις πόλεις που έχει επισκεφθεί κάθε μυρμήγκι δε θα πρέπει να γίνεται στο τέλος όλων των tour με memset που μηδενίζει όλο τον πίνακα για όλα τα μυρμήγκια. Αντίθετα κάθε μυρμήγκι θα μηδενίζει τη δική του πληροφορία και μόνο, μετά από ένα tour, με τη βοήθεια της συνάρτησης

void clearAntMask(int ant), ώστε να αποφύγουμε race conditions ανάμεσα στα threads

Βεβαιωνόμαστε για τη σωστή εκτέλεση του προγράμματος τρέχοντας τον κώδικα για λίγες πόλεις, όπως φαίνεται παρακάτω:



Στη συνέχεια εκτελούμε τον παράλληλο αλγόριθμο με χρήση των 4 thread του μηχανήματος για 10000 πόλεις με 4 μυρμήγκια και 10 γύρους. Η πολυπλοκότητα του αλγορίθμου είναι αρκετά μεγάλη και για τόσο μεγάλο αριθμό πόλεων και γύρων ο χρόνος εκτέλεσης είναι αρκετά μεγάλος, όπως φαίνεται παρακάτω:

Παρατηρούμε αρκετά καλή επιτάχυνση με τη χρήση 4 thread βλέποντας τους χρόνους real και user.

Ο **χρόνος απόκρισης** σε σχέση με την Εργασία 2 είναι σημαντικά μεγαλύτερος. Πιο συγκεκριμένα στην Εργασία 2 κυμαινόταν ανάμεσα στα **3-8sec** για 20000-50000 επαναλήψεις. Τώρα έχουμε **χρόνο 4min24sec** για 4 μυρμήγκια και 10 γύρους. Η συνολική απόσταση είναι **88731km,** σχεδόν ίδια με αυτή που υπολογίσαμε στην Εργασία 6 και είναι αρκετά καλύτερη από τα **3493734km** και **2848385km** της Εργασίας 2. Η μνήμη που χρησιμοποιήθηκε είναι ίδια με αυτή που χρησιμοποιείται στην Εργασία 6 και έχει αναλυθεί παραπάνω.

ΤΕΛΟΣ