Ειδικά Θέματα Παράλληλου Προγραμματισμού 2023-24

Σετ Ασκήσεων 1

Παναγιώτης Τριανταφυλλίδης

AM: 3200199

p3200199@aueb.gr

Το αντικείμενο της εργασίας είναι η φόρτωση και επεξεργασία εικόνων. Κάθε εικόνα αποτελείται από έναν αριθμό pixels όπου το χρώμα του κάθε πίξελ αναπαριστάται από τις τιμές των 3 βασικών χρωμάτων Red , Green , Blue . Συνήθως σε αυτά τα χρωματικά κανάλια προσθέτουμε και ένα τέταρτο κανάλι, Alpha , το οποίο αναπαριστά το ποσοστό της διαφάνειας του pixel. Τα κανάλια αποθηκεύονται ως μεταβλητές τύπου unsigned char , που παίρνουν τιμές από το 0 (καθόλου συνεισφορά) έως το 255 (μέγιστη συνεισφορά). Η κάθε εικόνα αποθηκεύεται ως ένας πίνακας από unsigned chars μεγέθους [width * height * number_of_channels].

Οι μετρήσεις έγιναν στο παρακάτω σύστημα:

Επεξεργαστής	AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx
Πλήθος πυρήνων	4
Πλήθος λογικών πυρήνων	8
Λειτουργικό σύστημα	Windows 10
Μεταφραστής	MSVC v143

Άσκηση 1

Το πρόβλημα

Μας δίνεται η συνάρτηση gaussian_blur_serial() , η οποία εφαρμόζει σειριακά το φίλτρο Gaussian Blur προκειμένου να θολώσει (ή να ομαλοποιήσει) μία εικόνα “garden.jpg”. Η συνάρτηση φορτώνει την εικόνα σε έναν πίνακα img_in, παράγει τη θολωμένη της εκδοχή img_out, βάσει μίας ακτίνας θόλωσης KERNEL_RADIUS (όσο μεγαλύτερη η ακτίνα, τόσο πιο έντονο το θόλωμα) και την αποθηκεύει σε ένα αρχείο τύπου JPG. Ζητείται να δημιουργήσουμε μία νέα συνάρτηση (gaussian_blur_parallel()), στην οποία να φορτώνεται η ίδια εικόνα και έπειτα να γίνεται η θόλωση παράλληλα, χρησιμοποιώντας είτε τα pthreads είτε τα C++ threads. Πρέπει να πειραματιστούμε με το μοίρασμα της δουλειάς σε 2, 4 και 8 threads και να χρονομετρήσουμε την απόδοσή τους και να τα συγκρίνουμε μεταξύ τους και με την σειριακή έκδοση. Τέλος να αποθηκεύσουμε την εικόνα σε ένα αρχείο με όνομα «blurred_image_parallel.jpg”

Μέθοδος παραλληλοποίησης

Χρησιμοποιήθηκε το σειριακό πρόγραμμα της gaussian_blur_serial για την δομή της της gaussian_blur parallel.

Για τον αρχικό ορισμό των νημάτων χρησιμοποιήθηκαν τα C++ Threads. Το πλήθος των νημάτων περνάει σαν όρισμα μαζί με την αρχική εικόνα. Ο ορισμός των νημάτων έγινε με την εντολή:

```
#include <thread>
#include <vector>

std::vector<std::thread> threads;
```

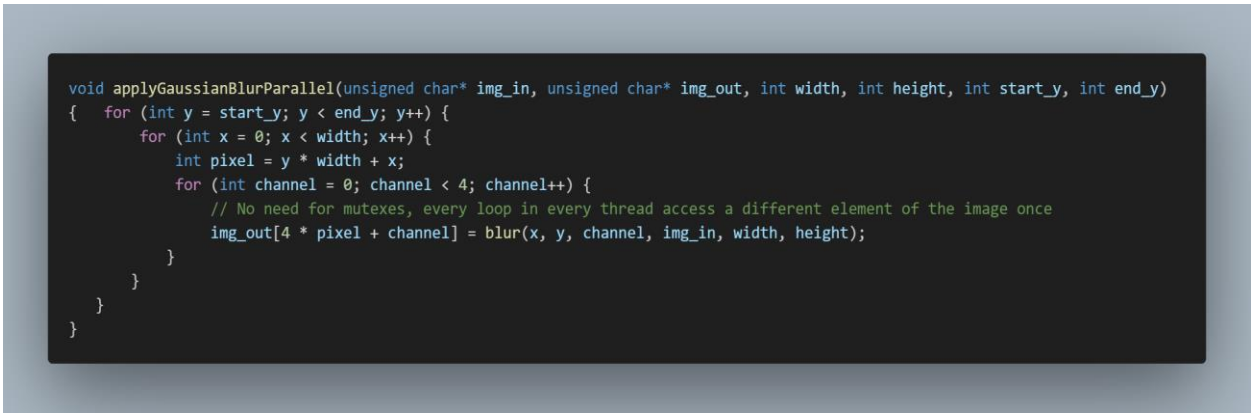
Κάθε νήμα αναλαμβάνει να εφαρμόσει σε ένα μέρος της εικόνας το Gaussian Blur φίλτρο, προκειμένου να θολώσει ένα τμήμα της. Το τμήμα της εικόνα το ορίζουν οι μεταβλητές start_y και end_y και για κάθε νήμα είναι εντελώς διαφορετικές οι τιμές. Έτσι κάθε νήμα θολώνει την εικόνα στον δικό του χώρο.

```
int band_height = height / num_threads;

// Timer to measure performance
auto start = std::chrono::high_resolution_clock::now();

for (int i = 0; i < num_threads; i++) {
      int start_y = i * band_height;
      int end_y = (i == num_threads - 1) ? height : start_y + band_height;
      threads.emplace_back(applyGaussianBlurParallel, img_in, img_out, width,
height, start_y, end_y);
                                                  ,
```

Κάθε νήμα εκτελεί την applyGaussianBlurParallel(). Για την παραλληλοποίηση εφαρμόζουμε το block_scheduling σαν τεχνική, όπου κάθε νήμα θολώνει ένα μέρος της εικόνας. Η επιλογή της εικόνας είναι κατάλληλη καθώς το ύψος της εικόνας (height == 2048) διαιρείται ακριβώς με το κάθε πλήθος νημάτων (2, 4, 8). Έτσι παραλληλοποιείται και ο πρώτος βρόχος. Παρακάτω δίνεται ο κώδικας:



Επιπλέον, είναι σημαντικό να σημειωθεί ότι δεν χρειάζεται να χρησιμοποιηθούν mutexes, καθώς κάθε επανάληψη από κάθε νήμα θολώνει διαφορετικό στοιχείο κάθε φορά, χωρίς να έχουμε κάποια σύγκρουση μεταξύ των νημάτων.

Τέλος, αφότου γίνει το join από τα νήματα, εγγράφεται από το αρχικό νήμα της main η τελική εικόνα.

Πειραματικά αποτελέσματα – μετρήσεις

Η χρονομέτρηση έγινε με τις εντολές :

```
// Timer to measure performance
auto start = std::chrono::high_resolution_clock::now();

auto end = std::chrono::high_resolution_clock::now();

// Computation time in milliseconds
int time = (int)std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count();
```

Τόσο στην σειριακή, όσο και στην παράλληλη έκδοση, χρονομετρήθηκε πριν την έναρξη της θόλωσης της εικόνας, μέχρι και το πέρας της εκτέλεσης της (πριν το γράψιμο στο τελικό αρχείο.

Το πρόγραμμα της σειριακής έκδοσης εκτελέστηκε 4 φορές και πάρθηκε ο μέσος χρόνος των εκτελέσεων. Αντίστοιχα εκτελέστηκε 4 φορές για 2, 4, και 8 νήματα.

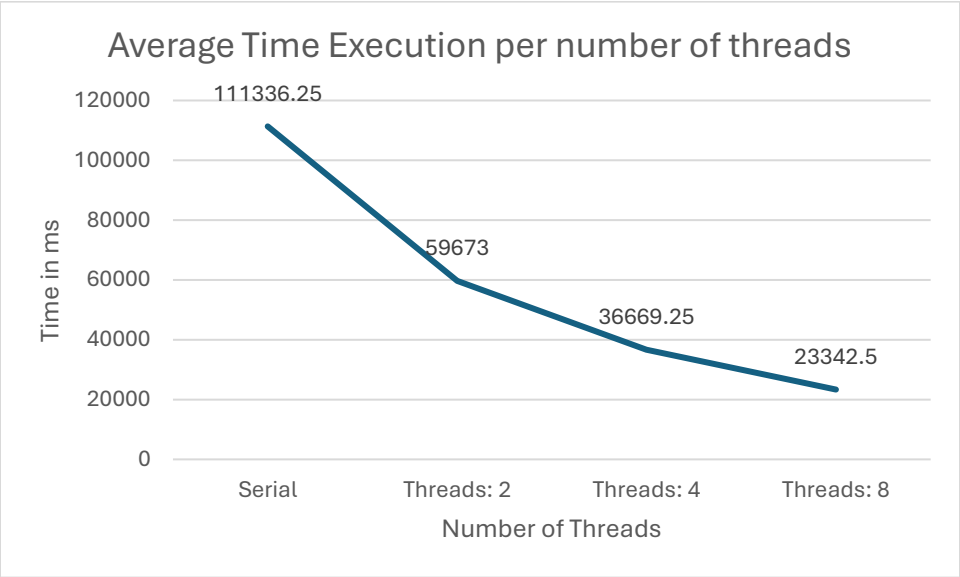
Οι μετρήσεις για κάθε μια από τις 4 εκτελέσεις είναι οι εξής:

Serial	**Threads: 2**	**Threads: 4**	**Threads: 8**
110626	58937	36828	24192
112251	60568	36567	24068
111178	59471	36568	22550
111290	59716	36714	22560

Και οι μέσοι χρόνοι είναι οι εξής:

Serial	**Threads: 2**	**Threads: 4**	**Threads: 8**
111336.25	59673	36669.25	23342.5

Διαγραμματικά:



Σχόλια

Από τους παραπάνω πίνακες αλλά και από τα διαγράμματα παρατηρούμε ότι όσο αυξάνεται το πλήθος νημάτων, τόσο μειώνεται ο συνολικός χρόνος εκτέλεσης. Συγκεκριμένα από 111.33625 sec σειριακά, μειώθηκε στα 59.673 sec για δύο νήματα, δηλαδή κοντά στο ½. Έπειτα για 4 νήματα έφτασε στα 36.66925 sec ενώ για 8 νήματα στα 23.3425 sec. Δηλαδή με την δημιουργία 8 νημάτων φτάσαμε κοντά στο 1/10 του αρχικού χρόνου.

Όλα τα παραπάνω είναι αναμενόμενα διότι κυρίως οι επαναλήψεις μοιράστηκαν ισόποσα, τόσο σε τεχνικό, όσο και σε λογικό επίπεδο. Για τα 4 νήματα, η μηχανή διαθέτει 4 cores, οπότε ωφέλησε προφανώς η αξιοποίηση τους για την επιτάχυνση του προγράμματος. Τώρα για τα 8 νήματα, η μηχανή διαθέτει 8 logical cores και αυτό σημαίνει ότι κάθε physical core μπορεί να εκτελέσει πολλά tasks ταυτόχρονα. Για αυτό παρατηρούμε ακόμα μια μεγαλύτερη μείωση με τα 8 νήματα, αφού κάθε core μπορεί να εκτελεί 2 threads ταυτόχρονα. Ουσιαστικά για 8 νήματα καταφέρνουμε να αξιοποιήσουμε όλους τους πυρήνες της μηχανή μας. Τέλος, η εκτέλεση του προγράμματος σε παράλληλο επίπεδο είναι άμεσα πιο γρήγορη, καθώς δεν έχουμε mutexes και δεν υφίσταται αργοπορία από σε τέτοιο επίπεδο.

Άσκηση 2

Το πρόβλημα

Μας δίνεται η συνάρτηση gaussian_blur_separate_serial() η οποία εφαρμόζει την τεχνική δύο περασμάτων από Gaussian Blur ώστε να επιταχύνει το αποτέλεσμα της θόλωσης της εικόνας “street_night.jpg” (Προσοχή : Το κάθε πέρασμα βασίζεται στο αποτέλεσμα του προηγούμενου περάσματος). Μας ζητείται να δημιουργήσουμε, με βάση την σειριακή συνάρτηση, μια νέα συνάρτηση με όνομα gaussian_blur_separate_parallel(). Η συνάρτηση θα φορτώνει την ίδια εικόνα και θα δημιουργεί 4 threads, τα οποία θα είναι ενεργά έως το τέλος του προγράμματος.

Μέθοδος παραλληλοποίησης

Η gaussian_blur_separate_parallel() χωρίζεται σε συγκεκριμένα σημεία, κρίσιμα για την παραλληλοποίηση του προγράμματος.

Αρχικά κάθε νήμα αναλαμβάνει τον υπολογισμός της μέγιστης τιμής των 4 channels ξεχωριστά, από όλα τα pixel της εικόνας. Δηλαδή κάθε νήμα αναλαμβάνει τον υπολογισμό της μέγιστης τιμής για κάθε κανάλι. Η συγκεκριμένη συνάρτηση που αναλαμβάνει την εκτέλεση είναι η εξής:

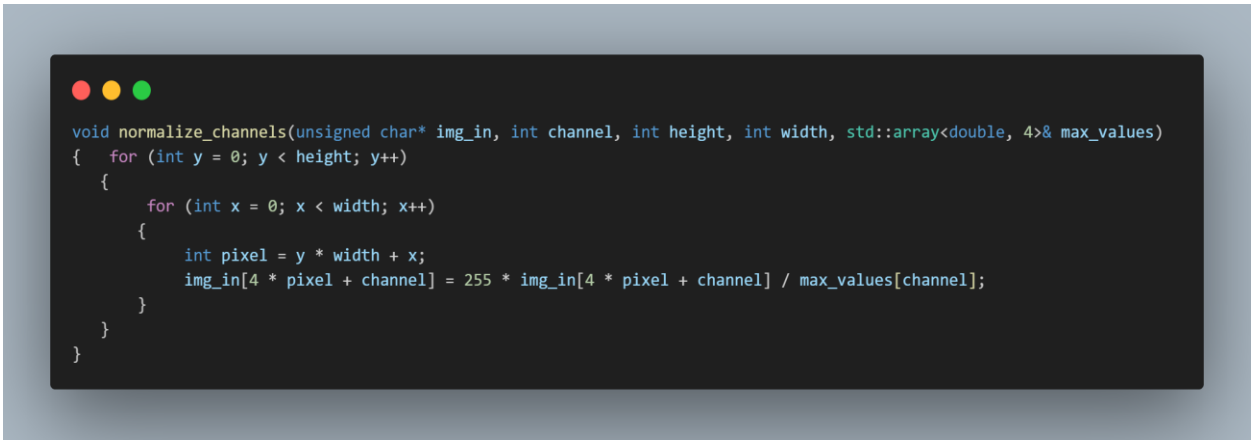
```
void calculate_max_value_channel(unsigned char* img_in, int channel, int height, int width, std::array<double, 4>& max_values)
{   double max_value = 0.0;
    for (int y = 0; y < height; y++)
   {
        for (int x = 0; x < width; x++)
       {
            int pixel = y * width + x;
            max_value = max_value < img_in[4 * pixel + channel] ? img_in[4 * pixel + channel] : max_value;
       }
   }
    max_values[channel] = max_value;
```

Στην οποία παρατηρούμε ότι κάθε νήμα εκτελεί τον ενδιάμεσο βρόχο για το δοσμένο κανάλι.

Έπειτα ακολουθεί η νορμαλοποίηση της εικόνας. Κάθε νήμα, δοσμένου του καναλιού, ομαλοποιεί την εικόνα με βάση τον τύπο:

```
pixel[channel] = 255*pixel[channel]/maxValue
```

Την νορμαλοποίηση την αναλαμβάνει η normalize_channels(), καλούμενη από κάθε νήμα:



```
void normalize_channels(unsigned char* img_in, int channel, int height, int width, std::array<double, 4>& max_values)
{   for (int y = 0; y < height; y++)
    {
        for (int x = 0; x < width; x++)
        {
            int pixel = y * width + x;
            img_in[4 * pixel + channel] = 255 * img_in[4 * pixel + channel] / max_values[channel];
        }
    }
}
```

Έπειτα, πρέπει να εγγραφεί η κανονικοποιημένη εικόνα από ένα thread σε ένα αρχείο με όνομα “image_normalized.jpg”. Για να εξασφαλίσουμε ότι ένα thread θα εκτελέσει μονάχα το γράψιμο αυτό, επικαλούμαστε τα atomic variables και συγκεκριμένα:

```
std::atomic<bool> task_write_normalized_image_done(false);
```

Όπου με την εντολή .exchange() εγγυόμαστε ότι ένα μόνο νήμα θα εκτελέσει την εγγραφή:

```
if (!task_write_normalized_image_done.exchange(true)) {
      stbi_write_jpg("image_normalized.jpg", width, height, 4/*channels*/,
img_in, 90 /*quality*/);
                                          }
```

Ωστόσο προσοχή, επειδή θα μπει το 1ο νήμα που θα είναι διαθέσιμο μετά την εκτέλεση της normalize_channels(), απαιτείται να χρησιμοποιήσουμε barrier ώστε να εξασφαλίσουμε ότι θα έχει τελειώσει η κανονικοποίηση της εικόνας και από τα 4 νήματα.

```
std::barrier sync_point(num_threads);
sync_point.arrive_and_wait();
```

Ακολούθως από την εγγραφή της εικόνας έπεται η θόλωση της εικόνας στον οριζόντιο άξονα. Σε αυτήν την περίπτωση δεν χρειάζεται να χρησιμοποιούμε barrier καθώς η εγγραφή της κανονικοποιημένης εικόνας γίνεται από την μεταβλητή img_in, ενώ η θόλωση εγγράφεται στην img_out, δηλαδή υπάρχει περίπτωση διαφορετικά νήματα να διαβάσουν αλλά να μην τροποποιήσουν την εικόνα.

Στην συνέχεια χρησιμοποιούμε το ίδιο barrier για να περιμένουμε να τελειώσουν τα νήματα, πριν εγγραφεί η θολωμένη οριζόντια εικόνα, αυτήν την φορά από την μεταβλητή img_out. Ο ορισμός της atomic variable είναι:

```
std::atomic<bool> task_write_horizontal_blurred_image_done(false);
```

Αντίστοιχα ο κώδικας είναι ο εξής:

```
// Only one thread is allowed to execute this task
if (!task_write_horizontal_blurred_image_done.exchange(true)) {
      stbi_write_jpg("image_blurred_horizontal.jpg", width, height,
4/*channels*/, img_out, 90 /*quality*/);
                                          }
```

Παρομοίως από την εγγραφή της εικόνας έπεται η θόλωση της εικόνας στον κάθετο άξονα. Σε αυτήν την περίπτωση δεν χρειάζεται να χρησιμοποιούμε barrier καθώς η εγγραφή της κανονικοποιημένης εικόνας γίνεται από την μεταβλητή img_out, ενώ η θόλωση εγγράφεται στην img_blur, δηλαδή υπάρχει περίπτωση διαφορετικά νήματα να διαβάσουν αλλά να μην τροποποιήσουν την εικόνα.

Για την παραλληλοποίηση της θόλωσης στην blur_parallel(), ακολουθήσαμε την ίδια τακτική με την άσκηση 1.

```
void blur_parallel(int start_y, int end_y, int width, int axis, unsigned char* img_in, unsigned char* img_blur) {

    for (int y = start_y; y < end_y; y++)
   {
        for (int x = 0; x < width; x++)
       {
            int pixel = y * width + x;
            for (int channel = 0; channel < 4; channel++)
           {
                img_blur[4 * pixel + channel] = blurAxis(x, y, channel, axis, img_in, width, end_y);
           }
       }
   }
}
```

Τέλος, η τελική εγγραφή της πλήρους θολωμένης εικόνας γίνεται από το νήμα της main και από την μεταβλητή img_blur.

Πειραματικά αποτελέσματα – μετρήσεις

Η χρονομέτρηση έγινε με τις εντολές :

```
// Timer to measure performance
auto start = std::chrono::high_resolution_clock::now();

auto end = std::chrono::high_resolution_clock::now();

// Computation time in milliseconds
int time = (int)std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count();
```

Τόσο στην σειριακή, όσο και στην παράλληλη έκδοση, χρονομετρήθηκε πριν την έναρξη της θόλωσης της εικόνας, μέχρι και το πέρας της εκτέλεσης της (πριν το γράψιμο στο τελικό άρχειο.

Το πρόγραμμα της σειριακής έκδοσης εκτελέστηκε 4 φορές και πάρθηκε ο μέσος χρόνος των εκτελέσεων. Αντίστοιχα εκτελέστηκε για την παράλληλη έκδοση των 4 νημάτων με τον ίδιο τρόπο.

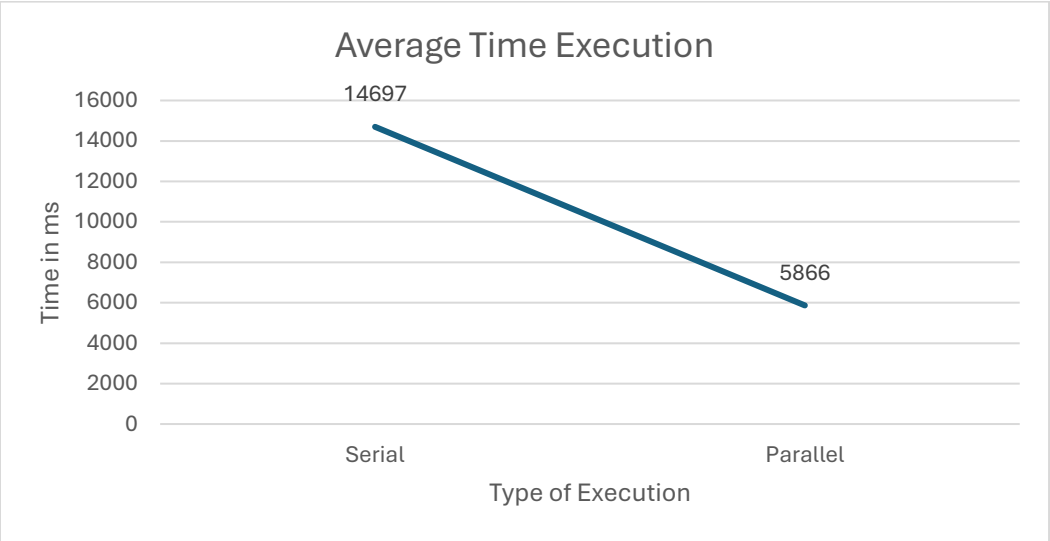
Οι μετρήσεις είναι οι εξής:

Serial	**Parallel**
14647	5837
14623	5979
14816	5852
14702	5796

Και οι μέσοι χρόνοι είναι οι εξής:

Serial	**Parallel**
14697	5866

Διαγραμματικά:



Σχόλια

Παρατηρούμε ότι ο μέσος συνολικός χρόνος από την σειριακή στην παράλληλη εκτέλεση γνώρισε σημαντική πτώση. Συγκεκριμένα, στην παράλληλη εκτέλεση έφτασε κοντά στο 1/3 της σειριακής εκτέλεσης. Αυτό είναι φυσιολογικό, καθώς τα 4 νήματα αναλαμβάνουν σημαντικό φόρτο εργασίας το καθένα. Ωστόσο, είναι σημαντικό να αναφερθεί πως στην παράλληλη έκδοση έχουμε κάποιες επιπλέον λειτουργίες που δεν εκτελούνται στην σειριακή, όπως:

- Εγγραφή της image_normalized
- Εγγραφή της image_blurred_horizontal
- Εγγραφή της image_blurred_final
- Κανονικοποίηση της εικόνας
- Εύρεση των μέγιστων κάθε καναλιού

Χωρίς αυτές τις λειτουργίες η εκτέλεση της παράλληλης έκδοσης θα ηταν ακόμα γρηγορότερη. Ωστόσο με τις δύο τελευταίες λειτουργίες καταφέρνουμε να έχουμε μια πιο καθαρή και φωτεινή εικόνα σε σχέση με την τελική θολωμένη της σειριακής εκτέλεσης. Επίσης, στην επιτάχυνση συμβάλλει και η ύπαρξη των img_blur και img_out, όπου χωρίς κάποια από αυτές θα χρειαζόταν να χρησιμοποιήσουμε barrier.

** *Στα συμπιεσμένα αρχεία συμπεριλαμβάνεται το βιβλίο εργασίας excel που περιέχει τις μετρήσεις και για τις 2 ασκήσεις, καθώς και screenshots της εκτέλεσης με το output των προγραμμάτων από το terminal της μηχανής στον φακελο resources. Η εκτέλεση έγινε σε περιβάλλον IDE: Microsoft Visual Studio και σε γλώσσα C++ στην έκδοση v20. Οι παραγόμενες εικόνες βρίσκονται στην ίδιο φάκελο με τον πηγαίο κώδικα.*