Lab 4: Concurrency

Part 1: FFI in Rust

Different programming languages have different strengths and weaknesses. Therefore, when working on a big project with various components, you may find using a particular language fits in some tasks since its strengths are more suitable for the given task. This is when FFI comes handy!

A foreign function interface (FFI) is a mechanism by which a program written in one programming language can call routines or make use of services written in another. For this reason, Rust provides FFI so it can interact with the other parts of the world such as C/C++, Ruby, Python, Node.js, C#,...

Rust has a keyword, `extern`, that facilitates the creation and use of a Foreign Function Interface. For example, consider the following code:

```
#[no_mangle]
pub unsafe extern "C" fn compute_distance(p1: Point, p2: Point) -> f64 {
   compute_distance(p1, p2)
}
```

The extern keyword links to or imports external code. The extern keyword is used in two places in Rust. One is in conjunction with the crate keyword to make your Rust code aware of other Rust crates in your project.

```
extern crate rug;
```

The other use is in foreign function interfaces, in which extern is used to define external blocks and declare function interfaces that Rust code can use to call foreign code.

```
#[link(name = "my_c_library")]
extern "C" {
    fn my_c_function(x: i32) -> bool;
}
```

This code would attempt to link with libmy_c_library.so on unix-like systems and my_c_library.dll on Windows at runtime and panic if it can't find something to link to. Rust code could then use my_c_function as if it were any other unsafe Rust function. Working with non-Rust languages and FFI is inherently unsafe, so wrappers are usually built around C APIs.

Let us consider that we have the following struct

```
pub struct Point {
   x: i8,
   y: i8,
}
```

To compute the distance between two points p1, and p2, we can use the Euclidean distance as:

$$d(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Question 1: Write a compute_euclidean_distance function to compute the Euclidean distance between two points.

```
pub fn compute_euclidean_distance(p1: &Point, p2: &Point) -> f64
```

Question 2: Write a test function to test the `compute_distance` function.

Now, let us assume that we want to define a function to compute the Manhattan distance between two points as:

$$d(p_1, p_2) = |x_1 - x_2| + |y_1 - y_2|$$

A sample code of that function can be written as:

```
pub fn compute_manhattan_distance(p1: &Point, p2: &Point) -> i32 {

   let a_abs  = (p2.x as i32 - p1.x as i32).abs();
   let a_abs  = (p2.y as i32 - p1.y as i32).abs();

   (a_abs + a_abs)
}
```

Rust can interact with c functions such as *abs()* by calling this function as follows:

First, you need to declare a dependency on `libc` by adding it to the Cargo.toml as follows:

```
[dependencies]
libc = "0.2.67"
```

Then you can use C functions inside Rust as follows:

```
extern {
    pub fn abs(i: i32) -> i32;
}

fn main() {
    let mut x1:i32=-2;
    print!("x: {:?}\n", x1);
    unsafe {
      println!("abs(x): {:?}",abs(x1));
    }
}
```

The previous code should output the following result:

```
x: -2
abs(x): 2
```

The code simply loads the libc library, which provides all of the definitions necessary to easily interoperate with C code on each of the platforms that Rust supports. This includes type definitions (e.g. c_int), constants (e.g. EINVAL) as well as function headers (e.g. malloc).

As a result, the compute_manhattan_distance function can be rewritten as follows to use the abs function:

```
#[repr(C)]
pub struct Point {
   x: i8,
   y: i8,
}

pub fn compute_manhattan_distanceC(p1: &Point, p2: &Point) -> i32 {
   unsafe
   {
      let a_abs  = abs(p2.x as i32 - p1.x as i32);
      let a_abs  = abs(p2.y as i32 - p1.y as i32);
      (a_abs + a_abs)
   }
}
```

Question 3: Write a Rust function to compute the ChebyshevDistance as:

```
pub fn compute_chebyshev_distance(p1: &Point, p2: &Point) -> i32
```

The Chebyshev Distance is formally defined as:

$$d(p_1, p_2) = max(|x_1 - x_2|, |y_1 - y_2|)$$

Question 4: Apply FFI to rewrite the function `compute_chebyshev_distance` into `compute_chebyshev_distance_C`, the function should use the abs() function (https://docs.rs/libc/0.2.67/libc/fn.abs.html).

Question 5: Utilize the code above to create a program in Rust that asks the user to input the coordinates of two points and then prompt the user to input what kind of distance they want to calculate. According to the user's input, the program should call the appropriate function.

- **DEMO this deliverable to the lab instructor.**

Part 2: Applying Concurrency with Rayon

Consider the following code:

```
struct Person {
    age: u32,
}
fn main() {
    let v: Vec<Person> = vec![
        Person { age: 23 },
        Person { age: 19 },
        Person { age: 42 },
        Person { age: 17 },
        Person { age: 17 },
        Person { age: 31 },
        Person { age: 30 },
    ];
    let num_over_30 = v.iter().filter(|&x| x.age > 30).count() as f32;
    let sum_over_30: u32 = v.iter().map(|x| x.age).filter(|&x| x > 30).sum();
    let avg_over_30 = sum_over_30 as f32/ num_over_30;
    println!("The average age of people older than 30 is {}", avg_over_30);
}
```

Question 6: what is the output of the program?

Rayon is a data-parallelism library that makes it easy to convert sequential computations into parallel. It is lightweight and convenient for introducing parallelism into existing code. It guarantees data-race free executions and takes advantage of parallelism when sensible, based on work-load at runtime. For example, the ParallelIterator module (https://docs.rs/rayon/0.6.0/rayon/par_iter/index.html) implements a concurrent iterator-style interface that allows you to write parallel programs.

Question 7: Alter the previous program (Question 6) to use par_iter instead of iter.

Consider using the following code:

```
  let mut v: Vec<Person> = Vec::new();
   for i in 1..10000 {
      v.push(Person { age: i });
   }
```

to benchmark your program from Question 7.

Question 8: Report the benchmarking output. Consider comparing your program with the original code that just uses iter(). Does using par_iter make a difference?

Question 9: Benchmark both the program with different vector sizes (1000, 10000, 100000, and 1000000) and provide your comments.

- **DEMO this deliverable to the lab instructor.**

Part 3: Drawing Pixels Concurrently

Consider the following example:

```
#[macro_use]
extern crate error_chain;
extern crate image;
extern crate num;
extern crate num_cpus;
extern crate threadpool;

use image::{ImageBuffer, Pixel, Rgb};
use num::complex::Complex;
use std::sync::mpsc::{channel, RecvError};
use threadpool::ThreadPool;

error_chain! {
    foreign_links {
        MpscRecv(RecvError);
        Io(std::io::Error);
    }
}

fn wavelength_to_rgb(wavelength: u32) -> Rgb<u8> {
    let wave = wavelength as f32;

    let (r, g, b) = match wavelength {
        380...439 => ((440. - wave) / (440. - 380.), 0.0, 1.0),
        440...489 => (0.0, (wave - 440.) / (490. - 440.), 1.0),
        490...509 => (0.0, 1.0, (510. - wave) / (510. - 490.)),
        510...579 => ((wave - 510.) / (580. - 510.), 1.0, 0.0),
        580...644 => (1.0, (645. - wave) / (645. - 580.), 0.0),
        645...780 => (1.0, 0.0, 0.0),
        _ => (0.0, 0.0, 0.0),
    };

    let factor = match wavelength {
        380...419 => 0.3 + 0.7 * (wave - 380.) / (420. - 380.),
        701...780 => 0.3 + 0.7 * (780. - wave) / (780. - 700.),
        _ => 1.0,
    };

    let (r, g, b) = (
        normalize(r, factor),
        normalize(g, factor),
        normalize(b, factor),
    );
    Rgb::from_channels(r, g, b, 0)
}

fn julia(c: Complex<f32>, x: u32, y: u32, width: u32, height: u32, max_iter:
u32) -> u32 {
    let width = width as f32;
    let height = height as f32;
```

```
    let mut z = Complex {
        // scale and translate the point to image coordinates
        re: 3.0 * (x as f32 - 0.5 * width) / width,
        im: 2.0 * (y as f32 - 0.5 * height) / height,
    };

    let mut i = 0;
    for t in 0..max_iter {
        if z.norm() >= 2.0 {
            break;
        }
        z = z * z + c;
        i = t;
    }
    i
}

fn normalize(color: f32, factor: f32) -> u8 {
    ((color * factor).powf(0.8) * 255.) as u8
}

fn main() -> Result<()> {
    let (width, height) = (1920, 1080);
    let mut img = ImageBuffer::new(width, height);
    let iterations = 300;

    let c = Complex::new(-0.8, 0.156);

    let pool = ThreadPool::new(num_cpus::get());
    let (tx, rx) = channel();

    for y in 0..height {
        let tx = tx.clone();
        pool.execute(move || {
            for x in 0..width {
                let i = julia(c, x, y, width, height, iterations);
                let pixel = wavelength_to_rgb(380 + i * 400 / iterations);
                tx.send((x, y, pixel)).expect("Could not send data!");
            }
        });
    }

    for _ in 0..(width * height) {
        let (x, y, pixel) = rx.recv()?;
        img.put_pixel(x, y, pixel);
    }
    let _ = img.save("output.png");
    Ok(())
}
```

The code above generates an image by drawing a fractal from the Julia set (https://en.wikipedia.org/wiki/Julia_set) with a thread pool for distributed computation.

Question 10: Change the Julia set (i.e., rewrite the julia function) to draw the Mandelbrot set (https://en.wikipedia.org/wiki/Mandelbrot_set).

Question 11: Replace the threadpool crate with Rayon crate, what kind of changes that you needed to apply to make the code work?

Question 12: Again, rewrite the Mandelbrot set drawing problem using the rayon crate.

- **DEMO this deliverable to the lab instructor.**