Assignment 1 - Introduction to HPC

Table of Contents

Code

The following was written in the Rust programming language.

```
#![feature(non_ascii_idents)] // feature flag that for the usage of `α` as a
variable name

use std::time::Instant;

/// the following function enacts the algorithm as specified in the assignment
fn advection(N: usize, dt: f32) -> f32 {
    let now = Instant::now(); // a time instant used for benchmarking

    let tmax = 2.0;
    let nbstep = (tmax / dt) as isize;
    let xmin = 0.0;
    let xmax = 1.0;
    let dx = (xmax - xmin) / 2.0;

    let v = 1.0;
    let xc = 0.25;

    let α = v * dt/(2.0 * dx);

    let x: Vec<f32> = (0..=N+2)
                          .map(|i| xmin + (i as f32 - 1.0))
                          .collect();

    let u0: Vec<f32> = (0..=N+2)
                   .map(|i| -200.0 * (x[i] - xc).powi(2))
                   .collect();
```

```
    let mut u = u0.clone();
    let mut unew = u0.clone();

    for timestep in 1..=nbstep {
        let current_time = timestep as f32 * dt;

        for j in 1..=(N + 1) {
            unew[j] = u[j] - α * (u[j+1] - u[j-1] + 0.5 * (u[j+1] - 2.0*u[j] +
u[j-1]))
        }
        u = unew.clone();

        u[0] = u[N+1];
        u[N+2] = u[1]
    }

    // measuring and returing elasped time
    now.elapsed().as_secs_f32()
}

/// a main function similar to java's or c's main()
fn main() {
    // runs the calculations 10 tines in a loop and benchmarks for case 1
    let case_1_results = (0..10).map(|_| advection(103, 0.0009))
                                .fold(0.0, |sum, x| sum + x) / 10.0;

    // runs the calculations 10 tines in a loop and benchmarks for case 2
    let case_2_results = (0..10).map(|_| advection(1003, 0.00009))
                                .fold(0.0, |sum, x| sum + x) / 10.0;

    println!("Average time taken for Case 1: N = 103, dt = 0.0009 over 10 runs is
{} seconds", case_1_results);
    println!("Average time taken for Case 2: N = 1003, dt = 0.00009 over 10 runs
is {} seconds", case_2_results);
}
```

Results (Part 1)

When compiled with the `--release` flag (which signals the compiler to produce and optimised binary), the time taken is -

```
Average time taken for Case 1: N = 103, dt = 0.0009 over 10 runs is 0.00054672
seconds
Average time taken for Case 2: N = 1003, dt = 0.00009 over 10 runs is 0.038205974
seconds
```

Results (Part 2)

When compiled with the `--release` flag (which signals the compiler to produce and optimised binary), the time taken is -

```
Average time taken for Case 1: N = 103, dt = 0.0009 over 10 runs is 0.00060271774
seconds
Average time taken for Case 2: N = 1003, dt = 0.00009 over 10 runs is 0.04333093
seconds
```

Reproducing results (Part 1)

Via pre-compiled binaries

```
# to run with optimization level 0 (debug version)
git clone https://github.com/DhruvDh/advection.git
cd advection
./advection-debug

# to run with optimization level 3 (release version)
git clone https://github.com/DhruvDh/advection.git
cd advection
./advection-release
```

Compiling yourself

To compile this on your machine first ensure you have the Rustup toolchain for Rust installed on your machine, which can be done via running the following command on Linux -

```
curl https://sh.rustup.rs -sSf | sh
```

For other operating systems visit https://rustup.rs/ and follow the instructions for your OS.

And then -

```
# to run with optimization level 0 (debug version)
git clone https://github.com/DhruvDh/advection.git
cd advection
cargo run

# to run with optimization level 3 (release version)
git clone https://github.com/DhruvDh/advection.git
cd advection
cargo run --release
```

Julia Version

Code

```
function advection(N, dt)
    tmax = 2.0
    nbstep = (tmax / dt)
    xmin = 0.0
    xmax = 1.0
    dx = (xmax - xmin) / 2.0

    v = 1.0
    xc = 0.25

    α = v * dt/(2.0 * dx)

    x = map(i -> xmin + (i-1)*dx, 1:(N+3))
    u0 = map(i -> -200 * (x[i] - xc)^2, 1:(N+3))

    u = copy(u0)
    unew = copy(u0)

    for timestamp = 1:nbstep
        current_time = timestamp * dt

        for j = 2:(N+2)
            unew[j] = u[j] - α*(u[j+1] - u[j-1]) + 0.5*(u[j+1] - 2*u[j] + u[j-1])
        end

        u = copy(unew)
        u[1] = u[N+2]
        u[N+3] = u[2]
    end
end

using BenchmarkTools

case_1_time = @belapsed advection(103, 0.0009)
case_2_time = @belapsed advection(1003, 0.00009)

println("Case 1: N = 103, dt = 0.0009 took $case_1_time seconds")
println("Case 2: N = 1003, dt = 0.00009 took $case_2_time seconds")
```

Results

N = 103 is faster than the Rust version, N = 1003 is about the same.

```
Case 1: N = 103, dt = 0.0009 took 0.0003366 seconds
Case 2: N = 1003, dt = 0.00009 took 0.042614101 seconds
```