Universidad Veracruzna

Inteligencia Artificial

Análisis de Algoritmos

Actividad 3

Leonardo Flores Torres

20 de septiembre de 2022

Programar un algoritmo de su elección (no tan sencillo) y analizarlo de la siguiente forma:

1. Graficar el tiempo de ejecución en función de N,
2. sobre los mismos ejes graficar 2 cotas superiores y dos cotas inferiores,
3. repetir el punto 1 y 2 ejecutando el programa en otra computadora de distinto desempeño,
4. analizar los resultados y discutirlos. Escribir de la manera más completa las características de las 2 computadoras.

El algoritmo que se eligió fue computar un fractal, más específicamente, el ...

El extracto de código que se muestra a continuación es una representación del modo de trabajo que se lleva en `julia` usando el REPL[1], asemeja un ambiente de trabajo y ejecución de comandos en la terminal.

```
julia> ns = 10:10:1600;                    # valores que puede tomar N
julia> reps = 20;                          # numero de repeticiones
julia> timings = zeros(length(ns), 2);     # arreglo bidimensional
julia> for rep in 1:reps
           for (index, n) in enumerate(ns)
               time = @elapsed mf.fractalCMap(n, n, maxiter=100)
               if rep == 1
                   timings[index, 1] = n
               end
               timings[index, 2] += time
           end
       end
julia> timings[:,2] = timings[:,2] / reps; # promedio de tiempo
julia> timings = vcat([0 0], timings);     # agregar entrada extra para el tiempo cero
julia> nvalues = timings[:,1];             # lista de iteraciones
julia> time = timings[:,2];                # lista de tiempos
```

Para graficar el tiempo de ejecución se hicieron 20 repeticiones indicadas por `reps`, y se definió una variable `ns` para guardar el conjunto de valores que puede tomar $N = 10,\ 20,\ 30,\ \ldots,\ 1600$. La variable `timings` guarda en la

[1]REPL es un acrónimo para Read-Eval-Print loop.

primera columna el valor de N, mientras que en la segunda columna guarda el tiempo $t(N)$ que le toma al algoritmo computar el fractal. Por cada iteración del `loop` se suman los tiempos $t(N)$ a sus respectivas entradas, y al final toda la columna de tiempos se divide entre la cantidad de repeticiones `reps` lo que resulta en tiempos promedio $\tilde{t}(N)$. Se usaron los tiempos promedio debido a que si hay procesos en ejecución en los ordenadores pueden generar cambios en los tiempos de cómputo.

Los tiempos de ejecución en el ordenador `diannao` se pueden observar en la figura 1 donde las dos líneas continuas representan las cotas superiores, mientras que las líneas no continuas con las pertenecientes a las cotas inferiores. Se tuvo que utilizar un factor de escalamiento para los cuatro casos ya que los tiempos eran muy pequeños incluso para el caso en el que $N = 1600$ donde el tamaño del fractal corresponderia a una imagen de 1600×1600 pixeles con un número de iteraciones máximo de 100.

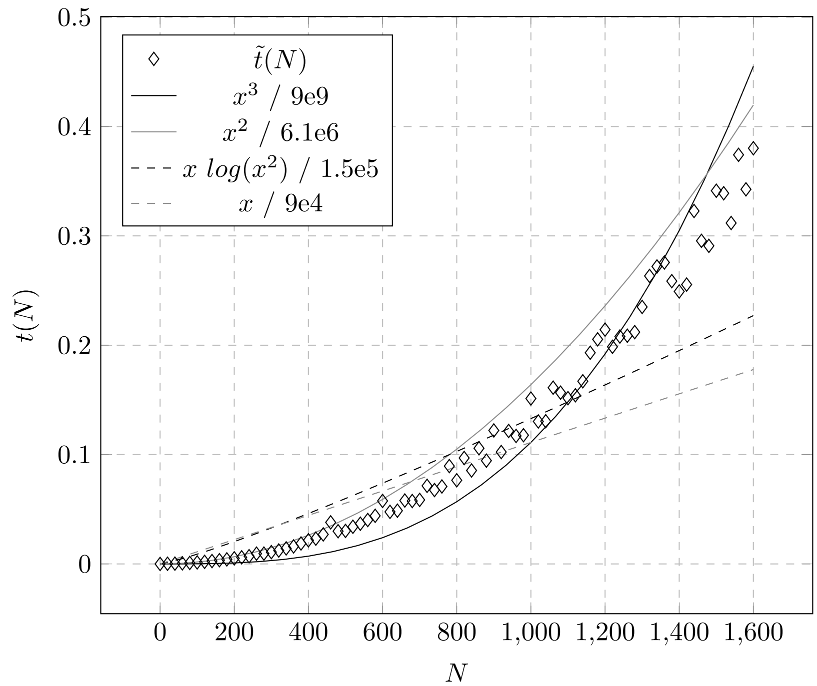

Figura 1: Tiempos de ejecución usando `hongdiannao`.

Las características de las computadoras usadas, `diannao` y `hongdiannao`, se muestran en la figuras 2 y 3, respectivamente.

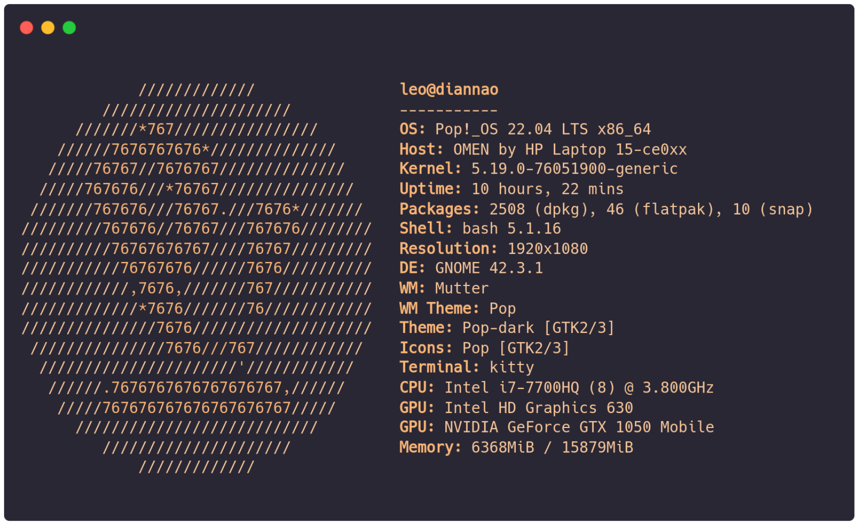

Figura 2: características del ordenador `diannao`.

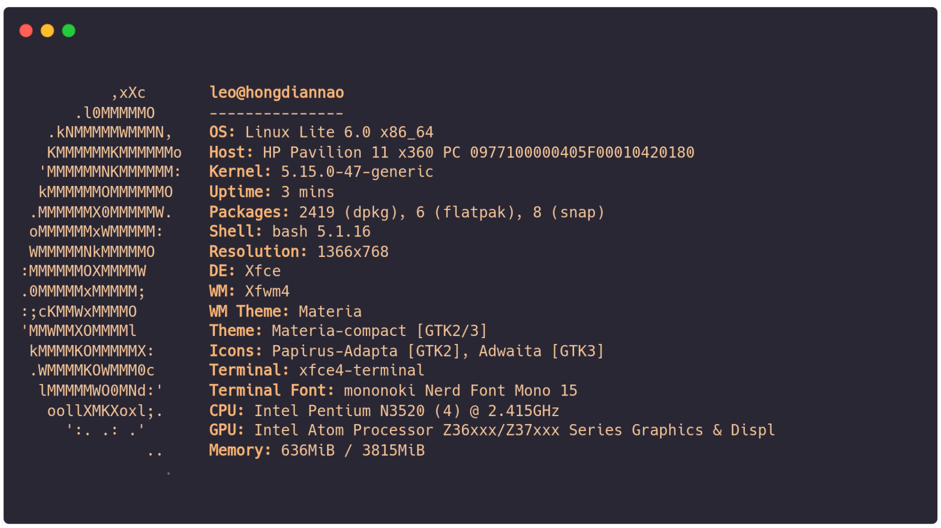

Figura 3: características del ordenador `hongdiannao`.

Apéndice

```julia
module MandelbrotFractal

#==============================
Required libraries
==============================#

using Images
using Plots

#==============================
Image operations
==============================#
```

```julia
function imageSave(path, img)
    save(path, img)
end

function visualize(rgbmap)
    return plot(rgbmap, ticks=false)
end

#==============================
Fractal genereration
==============================#

function mandelbrot(c, maxiter)
    z = 0
    n = 0
    while abs(z) <= 2 && n < maxiter
        z = z*z + c
        n += 1
    end
    return n
end

function canvasRGB(height, width)
    return zeros(RGB, height, width)
end

function canvasHSV(height, width)
    return zeros(HSV, height, width)
end

xc(i, lx, nx) = lx * (2*i - 1) / nx

function fractalGrays(nx, ny; lx=2, ly=2, maxiter=80)
    cv = canvasRGB(ny, nx)

    for row in 1:ny
        y = xc(row, ly, ny) - ly

        for col in 1:nx
            x = xc(col, lx, nx) - lx
            c = x + y * 1im
            m = mandelbrot(c, maxiter) / maxiter

            pixel = 1 - m |> RGB
            cv[row, col] = pixel
        end
    end
    return cv
end

function fractalColors(nx, ny; lx=2, ly=2, maxiter=80)
    cv = canvasHSV(ny, nx)

```

```
    for row in 1:ny
        y = xc(row, ly, ny) - ly

        for col in 1:nx
            x = xc(col, lx, nx) - lx
            c = x + y * 1im
            m = mandelbrot(c, maxiter) / maxiter

            hue = 360 * m           # H between 0 and 360 (color wheel)
            sat = 0.85              # S between 0 and 1 (saturation)
            val = m < 1 ? 1 : 0     # V between 0 and 1 (brightness)
            cv[row, col] = HSV(hue, sat, val)
        end
    end
    return cv
end

function fractalCMap(nx, ny; lx=2, ly=2, maxiter=80, cname="Oranges")
    cv = canvasRGB(ny, nx)

    cmap_divs = 200
    cmap = colormap(cname, cmap_divs, logscale=true) |> reverse

    for row in 1:ny
        y = xc(row, ly, ny) - ly

        for col in 1:nx
            x = xc(col, lx, nx) - lx
            c = x + y * 1im
            m = mandelbrot(c, maxiter) / maxiter

            # To select a color of the colormap
            cv[row, col] = cmap[ ceil(m * cmap_divs) |> Int ]
        end
    end
    return cv
end

end # module MandelbrotFractal
```