Universidad de los Andes

Facultad de Ingeniería
Maestría en Ingeniería de Sistemas
Concurrencia, Paralelismo y Distribución
Periodo 2024-10

Proyecto Paralelismo
Reporte de trabajo

Juan Diego Yepes Parra - 202022391
Esteban Gonzalez Ruales - 202021225
Felipe Núñez - 202021673

6 de Mayo de 2024
Bogotá D.C.

Índice

1. Introducción **2**

2. Explicación del programa principal **3**

3. Explicación de cada algoritmo **4**

3.1. Word Count 4

3.2. Image Rotation 6

1. Introducción

El presente trabajo tiene como objetivo desarrollar programas paralelos para ejecutarse en el clúster paralelo construido utilizando los dispositivos Raspberry Pi. Se abordarán dos tareas principales: contar palabras en un archivo de texto e implementar algoritmos de procesamiento de imágenes. Cada tarea requerirá el uso de técnicas de programación paralela para aprovechar eficazmente los recursos distribuidos del clúster.

Para la tarea de conteo de palabras, el programa se encargará de contar con precisión las ocurrencias de palabras en un archivo de texto, considerando reglas como la insensibilidad a mayúsculas y minúsculas, así como la puntuación. La implementación paralela será esencial para optimizar el rendimiento y la eficiencia del proceso de conteo en el clúster.

En cuanto al procesamiento de imágenes, el programa deberá leer y manipular imágenes, además de implementar el Método de Beier-Neely para la morfología de imágenes. Estas tareas requerirán técnicas de paralelización para distribuir el trabajo de manera eficiente entre los nodos del clúster y acelerar el procesamiento de imágenes.

Además, el proyecto incluirá mediciones de rendimiento y análisis para evaluar el comportamiento de los algoritmos en diferentes configuraciones de hardware. Estos análisis serán fundamentales para comprender el rendimiento y la escalabilidad de las soluciones implementadas en el clúster.

En resumen, el proyecto tiene como objetivo demostrar el potencial de la programación paralela para mejorar la eficiencia y el rendimiento de tareas de procesamiento de texto e imágenes en un entorno distribuido como el clúster de Raspberry Pi.

El código fuente que implementa esta tarea se puede encontrar en el siguiente link

https://github.com/juanyepesp/isis4218-parallelism-task

2. Explicación del programa principal

El programa principal despliega una serie de funcionalidades esenciales para la gestión dinámica y eficiente de un cluster de nodos. Una de estas funcionalidades consiste en la capacidad de agregar nodos adicionales al cluster de manera dinámica, permitiendo así una escalabilidad flexible del sistema. Mediante la función `connect_node`, se facilita la conexión con un nuevo nodo en caliente, sin interrumpir la operación del cluster. Esto garantiza una expansión fluida de la capacidad de procesamiento según sea necesario.

```
defp connect_node do
  node_name =
    IO.gets("\nEnter node name to connect to without double quotes or colon: ") |> String.trim()

  node_atom = :"#{node_name}"

  IO.puts("\nConnecting to node with atom: :#{node_atom} ...")

  case Node.connect(node_atom) do
    true -> IO.puts("Connected successfully")
    _ -> IO.puts("Error connecting to node")
  end

  show_connected_nodes()
end
```

Además, el programa principal proporciona herramientas para visualizar y administrar los nodos conectados al cluster. La función `show_connected_nodes` permite verificar los nodos actualmente conectados, lo que resulta fundamental para monitorear y gestionar la topología del cluster.

```
defp show_connected_nodes do
  IO.puts("\nNumber of nodes connected: #{Node.list() |> length}")
  IO.puts("Connected nodes:")
  IO.inspect(Node.list())
end
```

Otra funcionalidad crucial del programa es la capacidad de iniciar trabajadores remotos en los nodos secundarios del cluster. Esto se logra a través de la función `start_node_workers`, la cual despliega una instancia de ejecución en cada nodo secundario, listo para recibir comandos y datos para procesar. Esta capacidad de distribuir la carga de trabajo de manera equitativa entre los nodos del cluster mejora significativamente la eficiencia del procesamiento.

```
defp execution_loop do
  receive do
    {pid, msg} ->
      # IO.inspect("Received message: #{msg}")
      send(pid, {self(), :reply, "Message #{msg} received on pid #{inspect(self())}"})

```

```
      {pid, :count, list} ->
        # IO.inspect("")
        send(pid, {self(), :reply, remote_count(list)})

      {pid, :rotate, {chunk, width, height, angle, pixels}} ->
        send(pid, {self(), :reply, rotate_helper(chunk, width, height, angle, pixels)})

      {:kill} ->
        IO.puts("Killing worker with pid: #{inspect(self())}")
        exit(:normal)
    end

    execution_loop()
  end

  defp start_node_workers do
    node_pids =
      for node <- Node.list() do
        Node.spawn(node, fn -> execution_loop() end)
      end

    IO.puts("\nStarted remote workers on remote nodes with pids:")
    IO.inspect(node_pids)
    show_connected_nodes()
    node_pids
  end
```

Además de la capacidad de iniciar nodos, el programa principal también incorpora la funcionalidad de detenerlos según sea necesario. La función `stop_nodes` permite detener los nodos de manera controlada, lo que resulta útil para la gestión de recursos y la optimización del rendimiento del sistema.

```
    Enum.each(pids, fn pid -> send(pid, :kill) end)
```

En resumen, el programa principal actúa como una interfaz centralizada para la administración dinámica del cluster, facilitando la expansión, supervisión y gestión de los recursos de manera eficiente y flexible.

Por último se tienen las funcionalidades de contar las palabras y rotar una imagen, las cuales van a ser explicadas en las siguientes secciones.

3. Explicación de cada algoritmo

3.1. Word Count

El algoritmo para contar palabras consiste en contar todas las palabras de una entrada proporcionada. En el caso de nuestro grupo, hicimos que la entrada se leyera de un archivo de texto proporcionando el nombre del archivo. Cuando se ejecuta el algoritmo, este paraleliza internamente y une los resultados

para retornar un mapa unificado del conteo de las palabras dentro de la entrada.

Para paralelizar hay varios procesos intermedios del cual se debe saber. Inicialmente, el nodo principal, divide la entrada en chunks. El numero de chunks que se crean son el numero nodos que tiene el cluster para procesamiento. Creados los chunks, cada uno de estos es enviado a cada nodo para que cada uno de estos procese el chunk que se le envió.

```
defp count(text, pids) do
  words = String.split(text, ~r/[^a-zA-Z0-9']+/)
  pid_amount = length(pids)
  chunk_size = Float.ceil(length(words) / pid_amount) |> trunc()
  chunks = Enum.chunk_every(words, chunk_size)
  zipped = Enum.zip(pids, chunks)

  Enum.each(zipped, fn {pid, chunk} ->
    send(pid, {self(), :count, chunk})
  end)

  ...
end
```

Dentro de cada nodo, se realiza un proceso similar ya que cada nodo divide el chunk que se le envió en mini-chunks dependiendo del numero de núcleos que tenga el nodo. De esta manera, cada núcleo procesa un mini-chunk de palabras y cuenta la frecuencia de cada palabra. Cada uno de los núcleos retorna un mapa de acuerdo a lo que contó.

```
defp remote_count(list) do
  chunk_size = Float.ceil(length(list) / System.schedulers_online()) |> trunc()
  chunks = Enum.chunk_every(list, chunk_size)

  Task.async_stream(chunks, fn chunk ->
    count_helper(chunk)
  end)
  |> Stream.map(fn {:ok, result} -> result end)
  |> Enum.reduce(fn map1, map2 ->
    Map.merge(map1, map2, fn _, val1, val2 ->
      val1 + val2
    end)
  end)
end

defp count_helper(list) do
  list
  |> Stream.map(&String.downcase/1)
  |> Stream.reject(&(&1 == ""))
  |> Stream.reject(&String.starts_with?(&1, "'"))
```

```
    |> Stream.reject(&String.ends_with?(&1, "'"))
    |> Enum.group_by(fn x -> x end)
    |> Stream.map(fn {k, v} -> {k, Enum.count(v)} end)
    |> Map.new(fn {k, v} -> {k, v} end)
  end
```

Dado que cada núcleo devuelve un mapa individual, es necesario unificarlos, tarea de la cual se encarga cada uno de los nodos. Es decir, cada nodo se encarga de unificar los mapas de frecuencias de palabras que retorne cada núcleo. Habiendo unificado los mapas de los núcleos, cada uno de los nodos envía su mapa de frecuencias unificado al nodo principal donde se repite la misma tarea. El nodo principal recibe los mapas unificados de cada uno de los nodos y unifica todos en un solo mapa. Este mapa es el final y el que se retorna para que el usuario lo pueda ver.

```
defp count(text, pids) do
  ...

  Enum.reduce(1..pid_amount, %{}, fn _, acc ->
    receive do
      {_pid, :reply, result} ->
        Map.merge(acc, result, fn _, val1, val2 ->
          val1 + val2
        end)
    end
  end)
end
```

3.2. Image Rotation

El algoritmo para rotar las imágenes funciona de la siguiente manera. El proceso de rotación de imágenes se realiza mediante un algoritmo eficiente que garantiza la precisión y el rendimiento requeridos. Utilizando la librería Imagineer, la imagen se carga inicialmente. La función orquestadora recibe un mapa que codifica la imagen, permitiendo el acceso al bitmap a través del atributo `image.pixel`. Este bitmap se estructura como una lista de listas que representan cada píxel de la imagen.

Para acceder a cualquier atributo la imagen, se utiliza la llave del mapa `image` que representa el encoding de la imagen. Este mapa permite acceder al bitmap de la imagen, utilizando el atributo `image.pixels`. El proceso de rotación se fragmenta en `n` chunks, donde `n` corresponde al número de núcleos disponibles en el sistema. Cada chunk es sometido al algoritmo de rotación, el cual se detalla más adelante. Este algoritmo opera de forma independiente en cada fragmento de la imagen, permitiendo una ejecución paralela eficiente.

```
defp rotate_helper(chunk, width, height, angle, pixels) do
  # Calculate the sine and cosine of the angle
```

```
    sin_angle = Math.sin(angle)
    cos_angle = Math.cos(angle)

    # Center of rotation
    x0 = 0.5 * (width - 1)
    y0 = 0.5 * (height - 1)

    # Rotate the image. This can be done in parallel
    rotated_pixels =
      Enum.map(chunk, fn y ->
        Enum.map(0..(width-1), fn x ->
          a = x - x0
          b = y - y0
          xf = floor( + a * cos_angle - b * sin_angle + x0)
          yf = floor( + a * sin_angle + b * cos_angle + y0)

          pixel =
            if xf >= 0 && xf < width && yf >= 0 && yf < height do
              Enum.at(Enum.at(pixels, yf), xf)
            else
              {0, 0, 0}
            end
          {Kernel.elem(pixel, 0), Kernel.elem(pixel, 1), Kernel.elem(pixel, 2)}
        end)
      end)

    rotated_pixels
  end
```

A cada bitmap se le realizan las mismas operaciones. Se hace un recorrido, pixel por pixel, donde se calcula cuál pixel debe ir en la posición actual. Es decir, a través de una iteración pixel por pixel, se determina la posición de cada píxel tras la rotación. Esto implica calcular las coordenadas de rotación, considerando el ángulo especificado. En caso de que algún píxel se encuentre fuera de los límites de la imagen, se asigna un valor correspondiente a un píxel negro.

Finalmente, cada chunk que ya ha sido rotado se une en la función orquestadora. A continuación se muestra el código.

```
  defp rotate(image, angle, pids) do
    pid_amount = length(pids)

    # Define the rotation angle in radians
    deg = String.replace(angle, "\n", "") |> String.to_integer()
    angle = Math.deg2rad(deg)

    width = image.width
    height = image.height
    pixels = image.pixels
```

```
11
12    chunk_size = Float.ceil(height / pid_amount) |> trunc()
13    chunks = Enum.chunk_every(0..(height-1), chunk_size)
14    zipped = Enum.zip(pids, chunks)
15
16    Enum.each(zipped, fn {pid, chunk} ->
17      send(pid, {self(), :rotate, {chunk, width, height, angle, pixels}})
18    end)
19
20    # receive rotated chunks from workers
21    rotated_chunks = Enum.reduce(1..pid_amount, [], fn _, acc ->
22      receive do
23        {_pid, :reply, result} ->
24          [result | acc]
25      end
26    end)
27
28    # Reunite all of the rows
29    rotated_pixels = Enum.reduce(rotated_chunks, [], fn chunk, acc ->
30      acc ++ chunk
31    end)
32
33
34    rotated_image = %Imagineer.Image.PNG{
35        # ... more key value pairs
36      pixels: rotated_pixels,
37    }
38
39    rotated_image
40  end
```

Una vez que cada chunk ha sido rotado, se procede a reunir los resultados en la función orquestadora. Posteriormente, se crea un nuevo mapa `%Imagineer.Image.PNG`, que conserva todos los atributos de la imagen original, pero con el bitmap reemplazado por el resultado de la rotación. Este proceso garantiza la preservación de los metadatos y la información esencial de la imagen durante la rotación.