Estimación de π utilizando el Método de Monte Carlo Distribuido

Computación en Internet I

profesor: Milton Orlando Sarria

Jennifer Castro (A00400253)

Mateo Berrio (A00399877)

Juan Manuel Casanova (A00400090)

Juan Camilo Corrales (A00366910)

Método de Monte Carlo

El método de Monte Carlo es una técnica de simulación utilizada para resolver problemas matemáticos y físicos mediante el uso de aleatoriedad y estadística. Se basa en la generación de múltiples escenarios aleatorios para estimar resultados y evaluar la incertidumbre en modelos complejos.

El método implica la creación de una gran cantidad de muestras aleatorias de las variables de interés. Cada muestra se evalúa para obtener un resultado, y al repetir este proceso un número considerable de veces, se obtiene una distribución de resultados. Esta distribución permite calcular promedios, varianzas y otras estadísticas que ayudan a entender el comportamiento del sistema modelado.

El proceso general se puede resumir en los siguientes pasos:

1. **Definición del problema**: Identificar el sistema o problema que se desea analizar.
2. **Modelado**: Establecer un modelo matemático que represente el sistema, incluyendo variables aleatorias.
3. **Generación de muestras**: Utilizar generadores de números aleatorios para simular diferentes escenarios del modelo.
4. **Evaluación**: Calcular los resultados para cada escenario.
5. **Análisis de resultados**: Utilizar técnicas estadísticas para interpretar los datos obtenidos y hacer inferencias sobre el comportamiento del sistema.

Modelo cliente-maestro-trabajadores utilizado

- **Cliente**: Interactúa con el sistema para enviar solicitudes de tareas al maestro. El cliente envía una solicitud al maestro sin esperar una respuesta inmediata. Esta solicitud se envía a través de un canal de comunicación proporcionado por ICE, que maneja la serialización y el transporte de datos. El cliente puede continuar con otras operaciones mientras espera los resultados.
- **Maestro**: Actúa como coordinador del sistema. Recibe las solicitudes de los clientes, distribuye las tareas a los trabajadores y gestiona los resultados. Una vez que el maestro recibe la solicitud del cliente, puede asignar la tarea a uno o varios trabajadores de manera asíncrona. Esto significa que el maestro puede seguir procesando otras solicitudes sin esperar que los trabajadores completen la tarea.
- **Trabajadores**: Son instancias que realizan las tareas asignadas por el maestro. Los trabajadores, al finalizar sus tareas, envían los resultados de vuelta al maestro de manera asíncrona.

Estrategia de distribución de las tareas entre los trabajadores

1. Registro de Trabajadores

El maestro (MasterImplementation) tiene un método registerWorker que permite a los trabajadores registrarse a sí mismos. Cada trabajador crea un proxy (WorkerServicePrx) y lo envía al maestro, que lo añade a una lista (workerProxies). Esto permite al maestro tener conocimiento de todos los trabajadores disponibles para recibir tareas.

2. División de Tareas

Cuando se invoca el método estimatePi del maestro, se recibe el número total de puntos a utilizar en la estimación. El maestro determina cuántos trabajadores están registrados y calcula cuántos puntos le corresponderán a cada trabajador:

```
int pointsPerWorker = numPoints / numWorkers;
```

Esto significa que la carga de trabajo se distribuye de manera uniforme, dividiendo el total de puntos entre los trabajadores disponibles.

3. Asignación Asíncrona de Tareas

Para cada trabajador, se invoca el método calculatePointsAsync, que permite realizar la tarea de manera asíncrona. Esto se logra utilizando CompletableFuture, que permite que el maestro siga procesando mientras espera los resultados:

```
CompletableFuture<Integer> future =
worker.calculatePointsAsync(pointsPerWorker);
```

Al hacerlo de esta manera, el maestro puede enviar múltiples solicitudes a los trabajadores simultáneamente, mejorando la eficiencia y aprovechando la paralelización.

4. Recopilación de Resultados

Después de enviar las solicitudes, el maestro espera a que todos los CompletableFuture se completen utilizando CompletableFuture.allOf. Esto permite que el maestro se bloquee de forma no bloqueante hasta que todos los trabajadores terminen de calcular sus puntos:

```
CompletableFuture<Void> allOf =
CompletableFuture.allOf(futures.toArray(new CompletableFuture[0]));
allOf.get();
```

Una vez que todas las tareas han finalizado, el maestro recopila los resultados de cada CompletableFuture y los suma para calcular el total de puntos dentro del círculo.

5. Cálculo Final

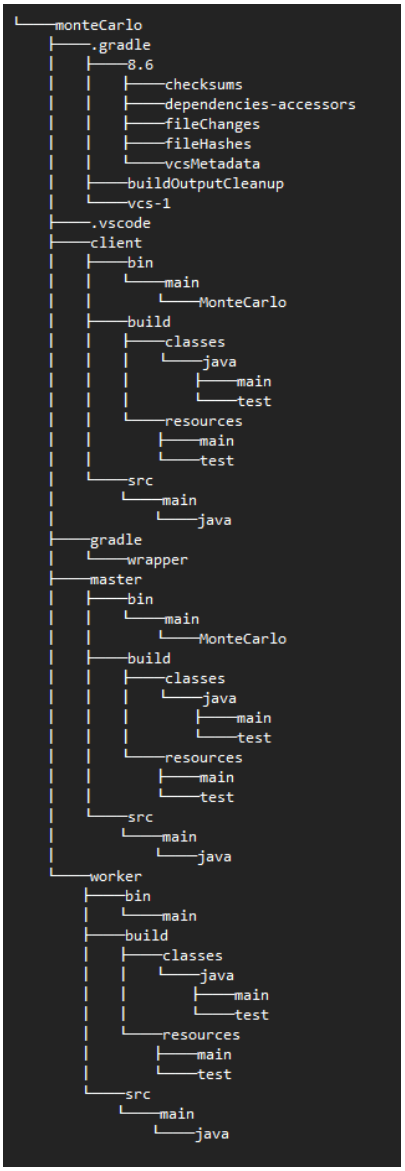
Finalmente, el maestro utiliza el total de puntos dentro del círculo para estimar el valor de π:

```
return 4.0f * totalPointsInCircle / numPoints;
```

De la manera que desarrollamos el codigo, se permite que múltiples trabajadores calculen puntos simultáneamente, minimizando los tiempos de espera y maximizando el uso de recursos.

Diseño e Implementación

Estructura del proyecto



Clase client

```java
public class Client {
    Run | Debug
    public static void main(String[] args) {
        try (Communicator communicator = com.zeroc.Ice.Util.initialize(args)){
            Scanner scanner = new Scanner(System.in);

            System.out.println(x:"Cliente iniciado...");
            System.out.println(x:"Escriba 'exit' para terminar o ingrese el número de puntos a calcular.");

            while (true) {
                System.out.print(s:"\nIngrese el número de puntos a calcular: ");
                String input = scanner.nextLine().trim();
                if (input.equalsIgnoreCase(anotherString:"exit")) {
                    System.out.println(x:"Finalizando cliente...");
                    break;
                }

                try {
                    int numPoints = Integer.parseInt(input);
                    if (numPoints <= 0) {
                        System.out.println(x:"Por favor, ingrese un número positivo de puntos.");
                        continue;
                    }
                    // Realizar la estimación de Pi sin especificar el número de workers
                    requestPiEstimation(communicator, numPoints).get();
                } catch (NumberFormatException e) {
                    System.out.println(x:"Por favor, ingrese un número válido o 'exit' para terminar.");
                } catch (Exception e) {
                    System.err.println("Error en la ejecución: " + e.getMessage());
                }
            }
        } catch (Exception e){
            e.printStackTrace();
        }
    }
```

```java
    public static CompletableFuture<Void> requestPiEstimation(Communicator communicator, int numPoints) {
        // Realizar la conexión con el servidor
        TaskCoordinatorPrx master = TaskCoordinatorPrx.checkedCast(
                communicator.stringToProxy("master:default -h localhost -p 10000"));

        if (master != null) {
            System.out.println("Solicitando estimación de π con " + numPoints + " puntos...");

            // Enviar solicitud de estimación sin especificar el número de workers
            return master.estimatePiAsync(numPoints)
                    .thenAccept(result -> {
                        System.out.println("\nResultados:");
                        System.out.println("------------");
                        System.out.println("Estimación de π: " + result);
                        System.out.println("Valor real de π: " + Math.PI);
                        System.out.println("Error absoluto: " + Math.abs(result - Math.PI));
                        System.out.println("------------");
                    });
        }
        return CompletableFuture.completedFuture(value:null);
    }

}
```

Clase Master

```java
public class Master {
    Run | Debug
    public static void main(String[] args) {
        try (Communicator communicator = com.zeroc.Ice.Util.initialize(args)) {
            // Crear un adaptador en el puerto 10000
            ObjectAdapter adapter = communicator.createObjectAdapterWithEndpoints("MasterAdapter",
                    "default -h localhost -p 10000");

            // Crear instancia del maestro
            Object MasterI = (Object) new MasterImplementation();

            // Registrar el objeto de maestro en el adaptador
            adapter.add(MasterI, com.zeroc.Ice.Util.stringToIdentity("master"));

            // Activar el adaptador
            adapter.activate();

            System.out.println(x:"El servidor maestro está listo...");
            communicator.waitForShutdown();
        }
    }
}
```

Clase Master implementation

```java
import MonteCarlo.WorkerServicePrx;
import com.zeroc.Ice.Communicator;
import com.zeroc.Ice.Current;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;

public class MasterImplementation implements TaskCoordinator{
    private List<WorkerServicePrx> workerProxies = new ArrayList<>();


    @Override
    // Método para registrar un worker directamente a través de su proxy
    public void registerWorker(WorkerServicePrx workerProxy, Current current) {
        if (workerProxy != null) {
            workerProxies.add(workerProxy);
            System.out.println("Worker registrado: " + workerProxy);
        } else {
            System.err.println(x:"Worker no válido.");
        }
    }

    @Override
    public float estimatePi(int numPoints, Current current) {
        int numWorkers = workerProxies.size();
        if (numWorkers == 0) {
            System.out.println(x:"No hay workers registrados para realizar la estimación.");
            return 0.0f;
        }

        System.out.println("Estimando el valor de pi con " + numPoints + " puntos y " + numWorkers + " trabajadores...");
        int pointsPerWorker = numPoints / numWorkers;

        List<CompletableFuture<Integer>> futures = new ArrayList<>();

```

```
        for (int i = 0; i < numWorkers; i++) {
            WorkerServicePrx worker = workerProxies.get(i);
            System.out.println("Solicitando al worker " + (i + 1) + " calcular " + pointsPerWorker + " puntos...");
            CompletableFuture<Integer> future = worker.calculatePointsAsync(pointsPerWorker);
            futures.add(future);
        }

        int totalPointsInCircle = 0;
        try {
            CompletableFuture<Void> allOf = CompletableFuture.allOf(
                    futures.toArray(new CompletableFuture[0])
            );

            allOf.get();

            for (CompletableFuture<Integer> future : futures) {
                totalPointsInCircle += future.get();
            }
        } catch (InterruptedException | ExecutionException e) {
            System.err.println("Error al obtener resultados: " + e.getMessage());
            return 0.0f;
        }

        return 4.0f * totalPointsInCircle / numPoints;
    }
}
```

Clase Worker

```
import MonteCarlo.WorkerServicePrx;
import MonteCarlo.WorkerService;
import com.zeroc.Ice.Communicator;
import com.zeroc.Ice.Object;
import com.zeroc.Ice.ObjectPrx;
import com.zeroc.Ice.ObjectAdapter;

public class Worker {
    Run | Debug
    public static void main(String[] args) {

        if (args.length < 1) {
            System.out.println(x:"Por favor, especifique el ID del trabajador como argumento.");
            return;
        }

        try (Communicator communicator = com.zeroc.Ice.Util.initialize(args)) {
            int workerId = Integer.parseInt(args[0]);

            MonteCarlo.WorkerService worker = new WorkerServiceI();

            // Crear un adaptador en el puerto correspondiente al worker ID
            ObjectAdapter adapter = communicator.createObjectAdapterWithEndpoints("WorkerAdapter",
                    "default -h localhost -p " + (10001 + workerId));

            ObjectPrx object = adapter.add(worker, com.zeroc.Ice.Util.stringToIdentity("worker" + workerId));
            adapter.activate();

```

```
            // Obtener referencia al master para registrarse
            TaskCoordinatorPrx master = TaskCoordinatorPrx.checkedCast(
                    communicator.stringToProxy("master:default -h localhost -p 10000"));

            if (master != null) {
                // Obtener proxy del worker para registrarse en el master
                WorkerServicePrx workerProxy = WorkerServicePrx.checkedCast(object);

                // Registrar el worker en el master
                master.registerWorker(workerProxy);
                System.out.println("Worker " + workerId + " registrado en el master.");
            } else {
                System.err.println(x:"No se pudo obtener el proxy del master. Asegúrate de que el master esté en ejecución.");
            }

            System.out.println("El trabajador " + workerId + " está listo y esperando tareas...");
            communicator.waitForShutdown();
        }
    }
}

```

Clase WorkerService

```
import com.zeroc.Ice.Current;

import java.util.Random;

public class WorkerServic  Random random  orkerService {
    private final Random random = new Random();

    @Override
    public int calculatePoints(int numPoints, Current current) {
        int pointsInsideCircle = 0;

        for (int i = 0; i < numPoints; i++) {
            // Generar puntos aleatorios entre -1 y 1
            double x = random.nextDouble() * 2 - 1;
            double y = random.nextDouble() * 2 - 1;

            // Verificar si el punto está dentro del círculo
            if (x * x + y * y <= 1) {
                pointsInsideCircle++;
            }
        }

        return pointsInsideCircle;
    }
}
```

Pruebas y Evaluación

Estimación con puntos diferentes

Objetivo: Evaluar la precisión de la estimación de π utilizando diferentes cantidades de puntos en el método de Monte Carlo. Esta prueba busca verificar que la estimación se mantenga dentro de una tolerancia aceptable conforme se aumenta el número de puntos.

```
   @Test
    public void testPiEstimationWithDifferentPoints() {
        int[] pointsValues = {1000, 10000, 100000, 1000000};

        for (int points : pointsValues) {
            double estimate = PiEstimator.estimatePi(points, 10);
            double error = Math.abs(estimate - PI_REAL);
            System.out.printf("N=%d puntos -> π estimado = %.10f, Error
= %.10f%n", points, estimate, error);
            assertTrue(error < TOLERANCE, "Error fuera de tolerancia
para N=" + points);
        }
    }
```

Metodología:

- Se definen un conjunto de valores que representan la cantidad de puntos (1000, 10000, 100000, 1000000).
- Para cada valor de puntos, se llama al método estimatePi de la clase PiEstimator utilizando 10 trabajadores.
- Se calcula el error absoluto entre la estimación obtenida y el valor real de π.
- Se utiliza assertTrue para confirmar que el error se encuentra por debajo de una tolerancia predefinida de 0.01.

Resultados Esperados:

- La estimación de π para cada cantidad de puntos debe ser tal que el error sea menor que 0.01. En caso de que alguna estimación exceda esta tolerancia, la prueba fallará, proporcionando información sobre el número de puntos utilizado.

Esta prueba es crucial porque asegura que el algoritmo de estimación de π mejora su precisión a medida que se incrementa el número de puntos, lo que es fundamental para validar la efectividad del método de Monte Carlo.

Varios trabajadores

Objetivo: Analizar el comportamiento del algoritmo de estimación de π cuando se utilizan diferentes configuraciones de trabajadores. Esto ayuda a comprender cómo la paralelización afecta la calidad de la estimación.

```
@Test
    public void testPiEstimationWithDifferentWorkers() {
        int points = 100000; // Fijamos un número de puntos para probar
distintos números de workers
        int[] workerCounts = {1, 2, 4, 8,12,13,14,15,15,17,18,19,20};

        for (int workers : workerCounts) {
            double estimate = PiEstimator.estimatePi(points, workers);
            double error = Math.abs(estimate - PI_REAL);
            System.out.printf("Workers=%d -> π estimado = %.10f, Error =
%.10f%n", workers, estimate, error);
            assertTrue(error < TOLERANCE, "Error fuera de tolerancia con
" + workers + " workers");
        }
    }
```

Metodología:

- Se fija un número de puntos en 100000 y se evalúan distintos números de trabajadores (1, 2, 4, 8, 12, 13, 14, 15, 17, 18, 19, 20).
- Para cada configuración, se llama al método estimatePi y se calcula el error absoluto en comparación con el valor real de π.
- Se aplica assertTrue para garantizar que el error sea inferior a 0.01.

Resultados Esperados:

- Se espera que cada estimación de π produzca un error que se mantenga dentro de la tolerancia. Si alguna configuración excede el umbral, la prueba fallará, indicando problemas en la estimación.

Esta prueba es esencial para evaluar la escalabilidad y eficiencia del algoritmo, garantizando que la paralelización no comprometa la calidad de la estimación, lo cual es crítico en aplicaciones que requieren rendimiento.

Puntos y trabajadores

Objetivo: Medir el tiempo de ejecución del algoritmo con diferentes combinaciones de puntos y trabajadores, evaluando así el rendimiento del sistema en función de la configuración.

```
   @Test
    public void testPerformanceWithDifferentWorkersAndPoints() {
        int[] pointsValues = {10000, 100000, 1000000};
        int[] workerCounts = {1, 2, 4, 8,12,13,14,15,15,17,18,19,20};

        for (int points : pointsValues) {
            System.out.printf("\n== Evaluación de rendimiento con N=%d
puntos ==%n", points);
            for (int workers : workerCounts) {
                long startTime = System.currentTimeMillis();
                double estimate = PiEstimator.estimatePi(points,
workers);
                long duration = System.currentTimeMillis() - startTime;
                System.out.printf("Workers=%d, N=%d -> Tiempo de
ejecución: %d ms, π estimado = %.10f%n",
                        workers, points, duration, estimate);
            }
        }
```

Metodología:

- Se define un conjunto de valores para la cantidad de puntos (10000, 100000, 1000000) y un conjunto para el número de trabajadores.
- Para cada combinación de puntos y trabajadores, se registra el tiempo que toma ejecutar el método estimatePi.
- Se imprime el tiempo de ejecución junto con la estimación de π.

Resultados Esperados:

- Se espera que los tiempos de ejecución se registren y que la estimación de π se mantenga razonablemente cerca del valor real, proporcionando datos sobre la eficiencia del algoritmo bajo diferentes configuraciones.

Esta prueba es crítica para la evaluación de la eficiencia y la escalabilidad del algoritmo. Permite identificar configuraciones óptimas para un rendimiento superior y ayuda a detectar cuellos de botella en la ejecución.

Evaluación del tiempo de ejecución

A continuación, presentamos una tabla en la que registramos el tiempo de ejecución que se tardó el programa en calcular la estimación de pi, comparándolo con el número de puntos y número de trabajadores. Los tiempos están dados en nanosegundos.

	Puntos									
Trabajadores	12	60	200	500	1000	1200	10000	60000	100000	600000
1	1	1	1	2	2	2	2	6	9	41
2	2	2	1	1	1	1	2	5	5	24
3	2	3	2	3	2	2	2	4	6	19
5	3	3	3	4	3	4	3	5	6	27
10	3	5	4	4	6	5	8	12	11	16

Se puede apreciar cómo a mayor número de trabajadores, el tiempo de ejecución es mucho mejor dada una cantidad de puntos muy grande. podemos ver que cuando son pocos puntos, muchos trabajadores se demoran un poco más que pocos trabajadores, y esto se debe a que cuando son pocos puntos, no se tiene tanta complejidad por lo que el programa tarda más en repartir el trabajo y hacer las funciones asíncronas, pero podemos apreciar que cuando se trata de muchos puntos, como son 600000, el rendimiento es mucho mejor, porque la complejidad es muy alta y ahí es donde la distribución del trabajo brilla.