

Instituto Politécnico do Cavado e Ave

Mestrado em Engenharia Informática

Programação Concorrente e Distribuída

2018/2019

***Fernando Correia a11199***

Conteúdo

[Introdução 3](#_Toc11941970)

# Introdução

# Resolução do problema

# Desenvolva uma aplicação sequencial (i.e., sem concorrência) para o cálculo de PI usando o método de Monte-Carlo que recebe como entrada a quantidade de pontos a gerar e produz como resultado o valor estimado de PI.

public static double formMontecarloSeq(long numeroPontos) {  
 for (long i = 1; i <= numeroPontos; i++) {  
 *x* = Math.*random*();  
 *y* = Math.*random*();  
 if (*x* \* *x* + *y* \* *y* <= 1)  
 *nSuccess*++;  
 }  
 return (4.0\* *nSuccess* / numeroPontos);  
}

Código para a estimativa de PI utilizando o método de Monte-Carlo sequencial.

Numero de pontos parametrizável no main method:

public static void main(String[] args) {  
 Scanner scannerObj = new Scanner(System.*in*); // Create a Scanner object  
 System.*out*.print("Informe a quantidade de iterações a calcular: ");  
 int numeroPontos = scannerObj.nextInt();  
 System.*out*.println("Nº Pontos : " + numeroPontos); // Output user input  
  
  
 System.*out*.println("Numero de processadores: " + Runtime.*getRuntime*().availableProcessors());  
 //Thread[] listathreads = new Thread[2];  
 long startTime = System.*nanoTime*();  
 double result = *formMontecarloSeq*(numeroPontos);  
 System.*out*.println("π: " + result);  
 long stopTime = System.*nanoTime*();  
 long totalTime = stopTime - startTime;  
 System.*out*.println("Calculo sequencial demorou (secs): " + String.*format*("%.6f", (totalTime)/1.0e9) );  
  
 long start = System.*currentTimeMillis*();  
 for (int i = 0; i < *ITERATIONS*; ++i) {  
 *formMontecarloSeq*(numeroPontos);  
 }  
 long elapsed = System.*currentTimeMillis*() - start;  
 long average = elapsed / *ITERATIONS*;  
 System.*out*.println("Para " + *ITERATIONS* + "iteracoes" + "o tempo médio foi: " + String.*format*("%.6f", (average)/1.0e9) + "s");  
}

# Desenvolva uma aplicação sequencial (i.e., sem concorrência) para o cálculo de PI usando o método de Séries de Gregory-Leibniz que recebe como entrada a quantidade de iterações k a calcular e produz como resultado o valor estimado de PI.

public static double formCarlzSeq(int numeroIteracoes) {  
 double factor = 1.0;  
 double sum = 0.0;  
 for (int k = 0; k < numeroIteracoes; k++) {  
 sum += factor / (2 \* k + 1);  
 factor = -factor;  
 }  
 return (4.0 \* sum);  
}

Código para a estimativa de PI utilizando o método de método de Séries de Gregory-Leibniz sequencial.

Numero de iterações parametrizável no main method:

public static void main(String[] args) {  
 Scanner scannerObj = new Scanner(System.*in*);  
 System.*out*.print("Informe a quantidade de iterações a calcular: ");  
 int numeroIteracoes = scannerObj.nextInt();  
 System.*out*.println("Nº Iterações : " + numeroIteracoes);  
  
  
 System.*out*.println("Gregory Numero de processadores: " + Runtime.*getRuntime*().availableProcessors());  
 long startTime = System.*nanoTime*();  
 double result = *formCarlzSeq*(numeroIteracoes);  
 long stopTime = System.*nanoTime*();  
 long totalTime = stopTime - startTime;  
 System.*out*.println("π: " + result);  
 System.*out*.println("Calculo Sequencial (secs): " + String.*format*("%.6f", (totalTime)/1.0e9) );  
  
 long start = System.*currentTimeMillis*();  
 for (int i = 0; i < *ITERATIONS*; ++i) {  
 *formCarlzSeq*(numeroIteracoes);  
 }  
 long elapsed = System.*currentTimeMillis*() - start;  
 long average = elapsed / *ITERATIONS*;  
 System.*out*.println("Para " + *ITERATIONS* + " iteracoes" + "o tempo médio foi: " + String.*format*("%.6f", (average)/1.0e9) + "s");  
}

# Apresente uma versão concorrente da aplicação com o método de Monte-Carlo, sendo que o programa deverá receber como entrada e produzir como resultado a mesma informação do programa sequencial. Adicionalmente, esta versão concorrente deve permitir também especificar como parâmetro de entrada o número de threads.

Para trabalhar com concorrência com este método usei os conceitos de “Atomic Integer”, “Workers” e “ExecutorService”.

Atomic Integers não são nada mais que um contador que se auto incrementa automaticamente.

@Override  
 public void run() {  
 double x = Math.*random*();  
 double y = Math.*random*();  
 if (x \* x + y \* y <= 1)  
 nAtomSuccess.incrementAndGet();  
 }  
}  
public MontecarloPar(int numeroPontos) {  
 this.nAtomSuccess = new AtomicInteger(0);  
 this.nThrows = numeroPontos;  
 this.value = 0;  
}

Se seguida usei a class “ExecuterService” para fazer a gestão de tarefas assíncronas. Uma vantagem é que podem ser terminadas a meio do processo.

public double getPi(int nProcessors) throws InterruptedException {  
 //int nProcessors = Runtime.getRuntime().availableProcessors();  
 ExecutorService executor = Executors.*newWorkStealingPool*(nProcessors);  
 Thread.*sleep*(3000);

Workers é um objecto que executa tarefas no background.

Runnable worker = new MonteCarlo();  
executor.execute(worker);

Código para receber como parâmetro o numero de threads a utilizar:

System.*out*.println("Nº threads : " + numetothreads);  
//ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(numetothreads);  
MontecarloPar PiVal = new MontecarloPar(numeroPontos);  
  
long startTime = System.*currentTimeMillis*();  
double value = PiVal.getPi(numetothreads);

# Apresente uma versão concorrente da aplicação com o método de Séries de Gregory-Leibniz, sendo que o programa deverá receber como entrada e produzir como resultado a mesma informação do programa sequencial. Adicionalmente, esta versão concorrente deve permitir também especificar como parâmetro de entrada o número de threads.

Para este problema decidir fazer a mesma abordagem do problema anterior mas partindo os blocos em quatro partes, ou seja, numa amostra de 2000000000 fiz:

//separa o cálculo em 4 partes definindo o valor de n inicial e final para cada uma  
Future<Double> parte1 = es.submit(new CarlzParalel(1, 300000000));  
Future<Double> parte2 = es.submit(new CarlzParalel(300000001, 600000000));  
Future<Double> parte3 = es.submit(new CarlzParalel(600000001, 900000000));  
Future<Double> parte4 = es.submit(new CarlzParalel(900000001, 1200000000));

***Qualquer um destes algoritmos está com os parâmetros que o professor pede no enunciado seja input de threads, amostras ou iterações tal como pode ver no código e nas explicações mais à frente deste relatório.***

# Analise do Desempenho

# De modo a medir o desempenho das implementações anteriores, acrescente no código das 4 aplicações anteriores um mecanismo de medição do tempo de execução

# durante a medição do tempo de execução não deve haver qualquer output para a consola

Todo o input é feito antes de ser feito qualquer calculo. Segue abaixo o exemplo do Monte-Carlo Paralelo:

Scanner scannerObj = new Scanner(System.*in*); // Create a Scanner object  
System.*out*.print("Informe a quantidade de iterações a calcular: ");  
int numeroPontos = scannerObj.nextInt();  
System.*out*.println("Nº Pontos : " + numeroPontos); // Output user input  
System.*out*.print("Informe a número máx de threads:");  
int numetothreads = scannerObj.nextInt();  
System.*out*.println("Nº threads : " + numetothreads);  
//ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(numetothreads);  
MontecarloPar PiVal = new MontecarloPar(numeroPontos);  
  
long startTime = System.*currentTimeMillis*();  
double value = PiVal.getPi(numetothreads);  
long stopTime = System.*currentTimeMillis*();

# o cálculo do tempo de execução na versão paralela deverá incluir a criação das threads.

Já respondido no código acima.

# a recolha do tempo de execução de cada aplicação deve ser uma média de várias execuções, configurável (por exemplo: 10), em vez de uma única execução.

Segue o exemplo de como está feito:

private final static int *ITERATIONS* = 10;

for (int i = 0; i < *ITERATIONS*; ++i) {  
 *formMontecarloSeq*(numeroPontos);  
}  
long elapsed = System.*currentTimeMillis*() - start;  
long average = elapsed / *ITERATIONS*;  
System.*out*.println("Para " + *ITERATIONS* + "iteracoes" + "o tempo médio foi: " + String.*format*("%.6f", (average)/1.0e9) + "s");

Optei por introduzir uma variável no programa para evitar mais inputs do utilizador. Está assim para os 4 problemas!

# Com base nos programas desenvolvidos nos pontos anteriores, construa uma tabela e um gráfico onde representa no eixo dos YY o tempo de execução do programa, e no eixo dos XX o nº de threads (assuma como exemplo: sequencial, PC1, PC2, PC4, PC8, PC10, PC20)

O valor escolhido para demorar pelo menos um minuto foi 2000000000. As durações são dadas em segundos.

Tal como já foi descrito em cima, Para Monte-Carlo Paralelo não usei os mesmo valores devido a problemas técnicos. Usarei o valor 40000000 para o numero de iterações do método Monte-Carlo com Paralelismo.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Número do Teste** | **Monte-Carlo Sequencial** | **Número de threads** | **Monte-Carlo Paralelo** | **Número de threads** | **Gregory-leibniz Sequencial** | **Número de Threads** | **Gregory-leibniz**  **Paralelo** | **Número de Threads** |
| 1 | 71,801922 | 12 |  | 1 | 2,635595 | 12 | 5,461960 | 1 |
| 2 | 73,216235 | 12 |  | 2 | 2,674446 | 12 | 4,014518 | 2 |
| 3 | 72,906707 | 12 |  | 3 | 2,653058 | 12 | 3,905925 | 3 |
| 4 | 71,633393 | 12 |  | 4 | 2,664836 | 12 | 1,945773 | 4 |
| 5 | 71,695355 | 12 |  | 5 | 3,851638 | 12 | 1,823980 | 5 |
| 6 | 72,842899 | 12 |  | 6 | 2,983383 | 12 | 1,885159 | 6 |
| 7 | 72,880306 | 12 |  | 7 | 2,878068 | 12 | 1,987230 | 7 |
| 8 | 71,809066 | 12 |  | 8 | 2,747102 | 12 | 2,197473 | 8 |
| 9 | 73,324869 | 12 |  | 9 | 2,690938 | 12 | 1,962592 | 9 |
| 10 | 72,312314 | 12 |  | 10 | 2,757905 | 12 | 1,937232 | 10 |
| 11 | 71,081354 | 12 |  | 11 | 2,656564 | 12 | 1,932596 | 11 |
| **12** | **72,429352** | **12** |  | **12** | **3,524187** | **12** | **1,953784** | **12** |
| 13 | 71,157345 | 12 |  | 1 | 2,779051 | 12 | 4,894508 | 1 |
| 14 | 73,478115 | 12 |  | 2 | 2,795472 | 12 | 3,584017 | 2 |
| 15 | 72,638695 | 12 |  | 3 | 2,915583 | 12 | 3,631574 | 3 |
| 16 | 72,999187 | 12 |  | 4 | 2,748540 | 12 | 1,938930 | 4 |
| 17 | 71,322452 | 12 |  | 5 | 3,004433 | 12 | 1,928179 | 5 |
| 18 | 71,197787 | 12 |  | 6 | 2,838585 | 12 | 1,925122 | 6 |
| 19 | 71,477868 | 12 |  | 7 | 3,220812 | 12 | 1,960427 | 7 |
| 20 | 72,447322 | 12 |  | 8 | 2,903680 | 12 | 2,321939 | 8 |
| 21 | 99,673832 | 12 |  | 9 |  |  | 1,944144 | 9 |
| 22 | 90,017828 | 12 |  | 10 |  |  | 1,930221 | 10 |
| 23 | 90,014973 | 12 |  | 11 |  |  | 1,953782 | 11 |
| **24** | **98,449943** | **12** |  | **12** |  |  | **1,953516** | **12** |

O Valor escolhido para estes valores foi 2000000000 iterações.