

Instituto Politécnico do Cavado e Ave

Mestrado em Engenharia Informática

Programação Concorrente e Distribuída

2018/2019

***Fernando Correia a11199***

Conteúdo

[Glossário 4](#_Toc12196637)

[Introdução 5](#_Toc12196638)

[Resolução do problema 6](#_Toc12196639)

[1. 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. 6](#_Toc12196640)

[2. 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. 7](#_Toc12196641)

[3. 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. 8](#_Toc12196642)

[4. MONTEParalel\_BETA 9](#_Toc12196643)

[5. 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. 9](#_Toc12196644)

[Analise do Desempenho 10](#_Toc12196645)

[1. De modo a medir o desempenho das implementações anteriores, acrescente no código das 4 aplicações anteriores um mecanismo de medição do tempo de execução 10](#_Toc12196646)

[ durante a medição do tempo de execução não deve haver qualquer output para a consola 10](#_Toc12196647)

[ o cálculo do tempo de execução na versão paralela deverá incluir a criação das threads. 10](#_Toc12196648)

[ 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. 10](#_Toc12196649)

[2. 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 11](#_Toc12196650)

[3. Compare o desempenho da execução concorrente com a execução sequencial, calculando o valor de Alfa (percentagem de código sequencial do programa) de acordo com a Lei de Amdahl, tomando como referência os ganhos de desempenho obtidos. Indique o tempo de execução medido para os casos sequencial e concorrente. Faça o cálculo para a melhor das versões concorrente (indicando o nº de threads assumido). 14](#_Toc12196651)

[• Indique o nº de processadores presentes na máquina usada para os resultados. 15](#_Toc12196652)

[4. Comente os resultados dos tempos de execução para as versões sequencial e concorrentes, nos dois métodos da aplicação, apresentando justificações para os valores obtidos. 16](#_Toc12196653)

# Glossário

1. **Speedup** – é uma medida do grau de desempenho. O speedup mede o rácio  
   entre o tempo de execução sequencial e o tempo de execução em paralelo.
2. **Lei de Amdahl** – dá-nos uma medida do speedup máximo que podemos obter ao  
   resolver um determinado problema com p processadores, também pode ser  
   utilizada para determinar o limite máximo de speedup que uma determinada  
   aplicação poderá alcançar independentemente do número de processadores a  
   utilizar.
3. **Thread** - é uma forma de um processo dividir a si mesmo em duas ou mais  
   tarefas que podem ser executadas concorrencialmente. O suporte à thread é  
   fornecido pelo próprio sistema operacional no caso da linha de execução ao  
   nível do núcleo KLT, ou implementada através de uma biblioteca de uma  
   determinada linguagem, no caso de uma ULT. Uma thread permite, por exemplo,  
   que o usuário de um programa utilize uma funcionalidade do ambiente  
   enquanto outras linhas de execução realizam outros cálculos e operações.
4. **Programação concorrente** - um paradigma de programação para a construção  
   de programas de computador que fazem uso da execução concorrente  
   (simultânea) de várias tarefas computacionais interativas, que podem ser  
   implementadas como programas separados ou como um conjunto de threads  
   criadas por um único programa. Essas tarefas podem ser executadas por um  
   único processador, vários processadores em um único equipamento ou  
   processadores distribuídos por uma rede.

# Introdução

Para a UC de Programação concorrente e distribuída do Mestrado em Engenharia Informática do Instituto Politécnico do Cavado e Ave, foi pedido pelo docente Nuno Lopes um trabalho para estimar o valor de PI pelos métodos Monte-Carlo e Gregory Leibniz através de uma grande quantidade de pontos / iterações de modo a poder obter resultados através de computação sequencial e paralela.

Segue em baixo todos os resultados.

# 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);

# MONTEParalel\_BETA

Segue neste trabalho uma tentativa de fazer uma outra abordagem concorrencial para o Método de Monte Carlo através de Executers e Futures de modo a poder partir o problema em várias partes e depois juntar o resultado. Está incompleto mas o objectivo passaria por conseguir meter mais inputs de forma a obter um resultado mais próximo.

# 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.

System.out.println("Nº threads : " + numetothreads);  
//https://stackoverflow.com/questions/949355/executors-newcachedthreadpool-versus-executors-newfixedthreadpool  
ExecutorService es = Executors.newFixedThreadPool(numetothreads);  
//separa o cálculo em 4 partes definindo o valor de n inicial e final para cada uma

# 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.

private final static int ITERATIONS = 10;

for (int i = 0; i <= ITERATIONS; ++i) {  
 parte1 = es.submit(new CarlzParalel(1, 500000000));  
 parte2 = es.submit(new CarlzParalel(500000001, 1000000000));  
 parte3 = es.submit(new CarlzParalel(1000000001, 1500000000));  
 parte4 = es.submit(new CarlzParalel(1500000001, 2000000000));  
}

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");

Como se pode ver por estes dois exemplos, o numero de ITERATIONS é um parâmetro parametrizável no inicio do programa.

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.

***Todos os resultados podem ser consultados no ficheiro Excel que vai dento do zip.***

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 e para manter coerência nos dados tenho duas tabelas, uma com o sequencial com 40000000 para poder fazer comparações e outra com 2000000000.

Ao tentar implementar o método Monte-Carlo paralelo com 2000000000 obtinha a seguinte mensagem de erro:

**Exception in thread "main" java.util.concurrent.RejectedExecutionException: Queue capacity exceeded**

**at java.util.concurrent.ForkJoinPool$WorkQueue.growArray(ForkJoinPool.java:884)**

**at java.util.concurrent.ForkJoinPool.externalSubmit(ForkJoinPool.java:2357)**

**at java.util.concurrent.ForkJoinPool.externalPush(ForkJoinPool.java:2419)**

**at java.util.concurrent.ForkJoinPool.execute(ForkJoinPool.java:2648)**

**at MontecarloPar.getPi(MontecarloPar.java:36)**

**at MontecarloPar.main(MontecarloPar.java:68)**

Tentei implementar o método Monte-Carlo Paralelo com a mesta técnica de divisão por partes mas infelizmente para o numero 2000000000 o intervalo de tempo de execução estava acima dos 1000s e não tive tempo para optimizar o algoritmo pelo que deixei noutro ficheiro como nome BETA. MONTEParalel\_BETA

***Calculo concorrencial (secs): 1096,989442***

**Para o método de Gregory-Leibniz temos os seguintes valores:**



Ter em atenção que Sequencial usou o número total de threads e paralelo usou sempre numero diferente de threads (1 a 12, e depois sempre 12). As conclusões tiradas é que depois de usar um número superior a 4 threads, os valores mantem-se constantes mas sempre decrescentes.

Para o método de Monte-Carlo temos os seguintes valores:



Os valores sequenciais para este resultado foram muito pequenos pelo que resolvi só implementar para o sequencial uma amostra maior de 2000000000.

Escolhi os valores máximos que me permitiram calcular ambos os casos sequencial e paralelo, infelizmente não consegui chegar a um minuto de tempo pois eu ficava sem memória antes ou a JVM gerava erro.

# Compare o desempenho da execução concorrente com a execução sequencial, calculando o valor de Alfa (percentagem de código sequencial do programa) de acordo com a Lei de Amdahl, tomando como referência os ganhos de desempenho obtidos. Indique o tempo de execução medido para os casos sequencial e concorrente. Faça o cálculo para a melhor das versões concorrente (indicando o nº de threads assumido).

Segue abaixo os resultados para ambos os métodos.

Speedup = (Valor Paralelo / Valor Sequencial)

Percentagem sequencial = ((nº cores / Speedup )- 1 / (nº cores-1))

*Para os cálculos da percentagem sequencial foram considerados o numero de threads usados no paralelismo, ou seja, o numero de threads usados nas sequencias paralelas.*

Eficiência Paralela = Speedup / nº cores execução paralela

Eficiência Sequencial = Speedup / nº cores execução sequencial

Tudo calculado no código

É possível verificar que no método Monte-Carlo seria possível meter mais pontos mas como não consegui resolver a questão da concorrência da melhor maneira, a JVM gerava um erro pelo que tive que introduzir os pontos máximos para conseguir executar o programa.

***Método Gregory-Leibniz***

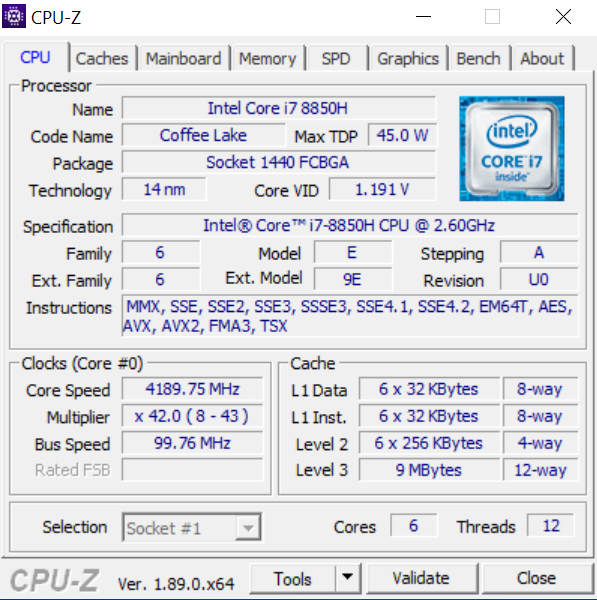


***Método Monte-Carlo***



# • Indique o nº de processadores presentes na máquina usada para os resultados.

Para estes resultados foi utilizada a seguinte configuração:



# Comente os resultados dos tempos de execução para as versões sequencial e concorrentes, nos dois métodos da aplicação, apresentando justificações para os valores obtidos.

Das vinte execuções das quais os resultados estão na tabela anterior assim  
como retratadas no gráfico acima.  
Podemos concluir que é de facto uma vantagem tirar partido da programação  
em paralelo isto é usar threads, e colocar cada uma a fazer tarefas  
independentes sendo que desta forma a rapidez de execução de um programa  
será de fato aumentada conforme podemos constatar pelos resultados obtidos.  
Podemos também ressalvar que criar um programa em paralelo e usar uma  
única threads, não faz sentido pois será mais lento do que o mesmo programa  
em sequencial e desta forma não sendo possível calcular o Alfa pois é  
impossível aplicar a Lei de Amdahl, visto que teríamos de dividir por zero sendo  
que isso não é matemática possível.  
O desenvolvido programa é determinístico, isto é, obtém-se sempre o resultado  
esperado.  
Constata-se que aumentado o número de threads a rapidez é maior obtendo-se  
assim um speedUp, ou seja, um ganho maior, conforme se pode constatar nos  
resultados obtidos. Não adianta juntar cores para obter maior velocidade de processos mas pode-se hoje em dia adoptar técnicas para melhorar os processos em execução.

Tive pena por só conseguir usar 40000000 no método Monte-Carlo pois acho que com valores superiores iria obter resultados mais interessantes!