

**VISÃO POR COMPUTADOR**

**Trabalho Prático de Programação Concorrente**

Ricardo Dias nº11597

Daniel Correia nº17497

21/06/2019

**Resumo**

Neste Trabalho Prático Programação Concorrente o desafio proposto pelo docente passava por desenvolver um programa em java que fosse capaz de calcular o valor aproximado de pi através de dois métodos diferentes sendo eles método de Monte-Carlo, e o método de Séries de Gregory-Leibniz, sendo que devem existir 2 versões diferentes para cada um dos métodos uma versão sequencial e uma versão concorrente.

Posteriormente será analisado o resultado obtido em ambas as versões.

**Índice**

[1. Introdução 5](#_Toc11705467)

[2. Desenvolvimento 7](#_Toc11705468)

[3. Conclusões 13](#_Toc11705469)

**Índice** **Figuras**

[Figura 1 – Função IsPar. 7](#_Toc11759385)

[Figura 2 - Ciclo sequencial Séries de Gregory-Leibniz 7](#_Toc11759386)

[Figura 3 – getPi. 8](#_Toc11759387)

[Figura 4 - Cálculo de Pi Sequencial Monte-Carlo 8](#_Toc11759388)

[Figura 5 – Método “run” versão concorrente de Séries de Gregory-Leibniz 9](#_Toc11759389)

[Figura 6 - Criação de threads 10](#_Toc11759390)

[Figura 7 - Metodo run do metodo de Monte-Carlo 10](#_Toc11759391)

1. Introdução

Neste Trabalho Prático o desafio proposto pelo docente passava por desenvolver um programa em java, capaz de calcular o valor aproximado de pi através de dois métodos diferentes sendo eles método de Monte-Carlo, e o método de Séries de Gregory-Leibniz ,alguns dos parâmetros necessários pelo docente são os de permitir ao utilizador introduzir através da linha de comandos o numero de iterações necessárias ou o número de pontos a ser gerado dependendo do método em causa, de notar que nas versões concorrentes é necessário também ter a possibilidade do utilizador selecionar o número de threads que deseja serem criadas.

Foi decidido pelo que grupo que o melhor seria dividir o código em quatro programas diferentes independentes uns dos outros, sendo dois para ambas as versões sequencias e dois para as respetivas versões concorrentes, outro ponto importante a desenvolver na primeira fase do trabalho é a utilização de técnicas de programação que garantam a correção do programa, necessário para produzir sempre o mesmo resultado determinístico.

E como último ponto será comparado o resultado obtido entre a versão sequencial e a versão paralela, necessário para a analise dos programas.

2. Desenvolvimento

Na primeira fase foram criadas ambas as versões sequenciais dos respetivos programas, ambas com a possibilidade de indicar o numero de interações ou o numero de pontos a serem gerados, denotar que na versão sequencial do programa método de Séries de Gregory-Leibniz o professor indicava no enunciado para ter em atenção ao possível conflito com a variável factor, e foi optado pelo grupo já na versão sequencial embora não sendo necessário ter em atenção este conflito e aplicar o algoritmo de forma a que este conflito seja tratado.

Para isso foi criado uma função IsPar que irá verificar se uma variável é par ou não e caso seja devolve “true” e vice-versa, esta função é mais tarde usada no ciclo que irá percorrer o numero de iterações especificadas e irá verificar para cada iteração se está é par ou não e conforme isso será mais tarde alterado o valor da variável factor para depois ser calculada a variável sum que multiplicada por 4 após o ciclo nos irá dar o valor aproximado de pi.

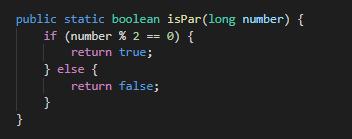


Figura 1 – Função IsPar.

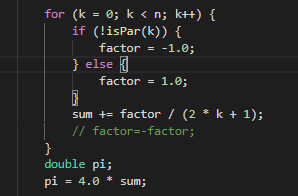


Figura 2 - Ciclo sequencial Séries de Gregory-Leibniz

Quanto à versão sequencial do programa que utiliza o método de Monte-Carlo, a função mais importante para o funcionamento deste é a função getPi, que tem como função percorrer um ciclo, que neste caso será o número de pontos a ser gerados introduzido pelo utilizador, e nesta função irá criar para cada ciclo 2 números aleatórios através da função “Math.random” e depois irá verificar se o quadrado da soma destes é menor ou igual a 1 e caso seja, irá incrementar uma varável de retorno da função, está variável de retorno é importantíssima pois diz-nos os pontos em que o raio da circunferência é menor ou igual a 1, pontos estes que serão mais tardes divididos pelo número de pontos a serem gerados e multiplicado por 4, o resultado obtido representa o nosso valor aproximado de PI.

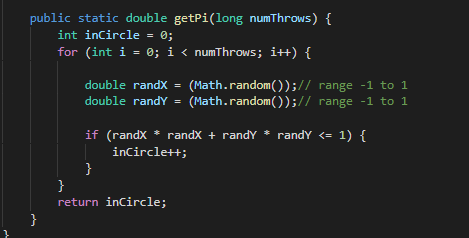


Figura 3 – getPi.

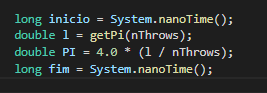


Figura 4 - Cálculo de Pi Sequencial Monte-Carlo

Quanto às versões concorrentes começando pelo método de Séries de Gregory-Leibniz, transportamos a função IsPar() para a função run(), função esta que será usada por todas as threads criadas, assim cada thread percorrerá um conjunto de números dependente das iterações e nº threads desejadas pelo utilizador, sendo que para o mesmo número de iterações quanto maior for o número de threads menor será o nº de blocos pertencente a cada thread, neste caso foi decidido guardar o valor de cada soma numa variável global total que irá somar o valor de soma existente com o calculado pela thread seguinte.

No main a forma de criação de threads foi igual para ambas as versões concorrentes, nas quais são criadas à medida que é percorrido o ciclo para o número de threads indicados pelo utilizador e para esse número é dividido uniformemente o valor de iterações indicado.

A grande diferença entre as diferentes versões trata-se do algoritmo implementado no método run embora o valor do sum é guardado numa variável global total. Foi utilizado em ambos um lock de forma a sincronizar e diminuir a possibilidade de erros nas variáveis globais quando é percorrido no run os respetivos ciclos para cada thread.

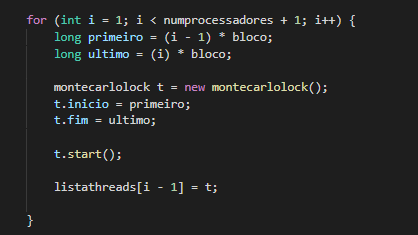


Figura 5 - Criação de threads

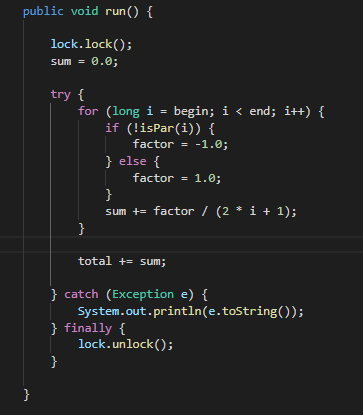


Figura 6 – Método “run” versão concorrente de Séries de Gregory-Leibniz



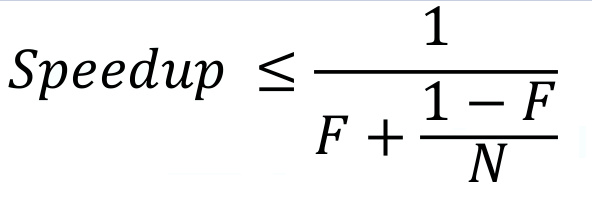
Figura 7 - Método “run” versão de Monte-Carlo

Quanto à parte de Análise de Desempenho de forma a cumprir os requisitos da parte 1) foi adicionado ao código existente duas variáveis que recolhem o tempo do sistema no inicio do programa antes da criação das threads e medido o tempo do sistema no final de calculado o valor de pi, depois é criada uma variável de tempo que trata-se da diferença de estes dois tempos de forma a conseguirmos fazer uma medição do tempo de execução do programa, este tempo é recolhido em segundos.

Análise de Desempenho

De forma a responder aos pontos 1,2,3 e 4 da secção da análise de desempenho resultante da comparação do desempenho da execução concorrente com a execução sequencial, é apresentada uma tabela e gráficos com os resultados obtidos e com o indicador de medida de acordo com a Lei de Amdahl (Eq 1).

Lei de Amdahl define o máximo ganho possível na execução de um programa:



Eq 1 - Lei de Amdahl.

F – Representa a fracção sequencial.

N- Número de núcleos.

Speedup – Representa a relação entre o [tempo](https://pt.wikipedia.org/wiki/Tempo) gasto para executar uma tarefa com um único processador e o tempo gasto com N processadores.

{\displaystyle S={T(1) \over T(N)}}

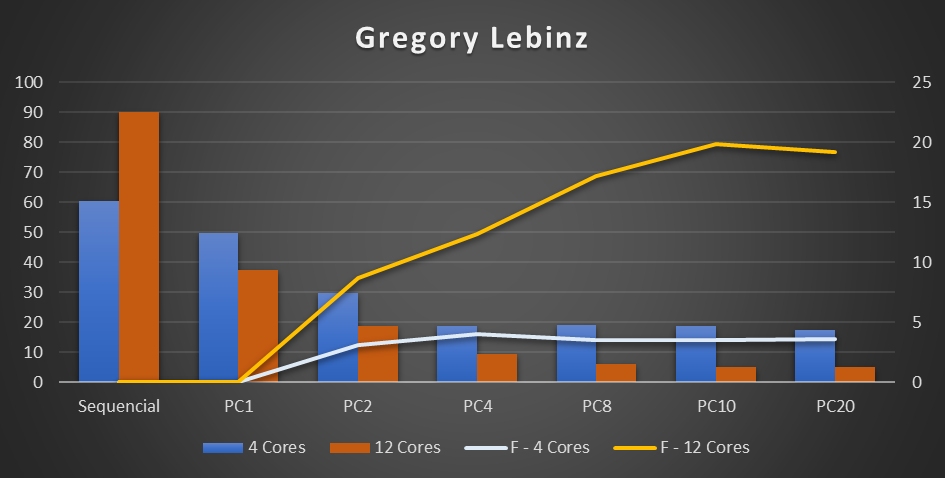


Foi também cálculado o erro em comparação do resultado com o valor de Pi do sistema.

**Gregory Lebinz**

|  |
| --- |
| NºIterações |
| 16000000000 |

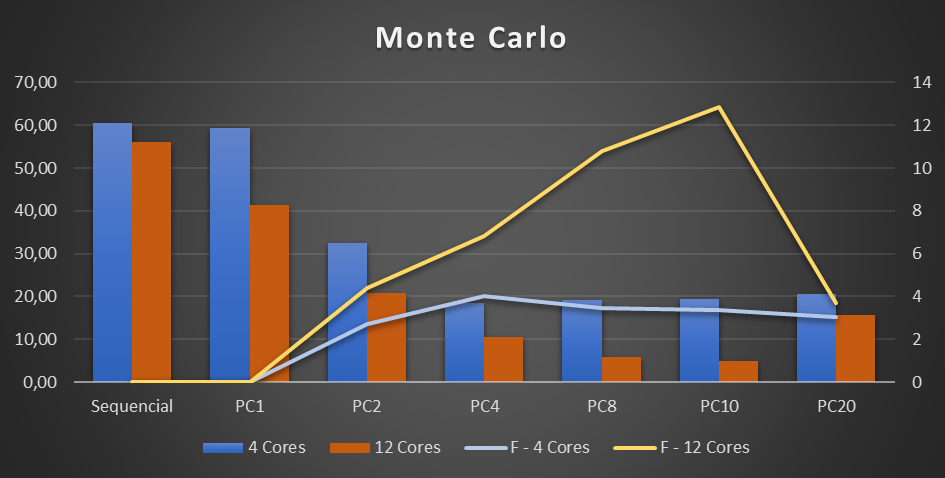
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Nº Núcleos | | Nº Núcleos | |  |
|  | 4 | 12 | 4 | 12 |  |
| NºThreads | Tempo(s) | Tempo(s) | F | F | Erro (pi - π) / π \* 100 |
| Sequencial | 60,38 | 90,15 | - | - | -2,03611E-09 |
| PC1 | 49,79 | 37,30 | - | - | -2,03611E-09 |
| PC2 | 29,72 | 18,68 | 3,06 | 8,65 | -2,03825E-09 |
| PC4 | 18,66 | 9,46 | 3,98 | 12,37 | -2,04845E-09 |
| PC8 | 18,87 | 5,93 | 3,51 | 17,23 | -2,04408E-09 |
| PC10 | 18,67 | 5,01 | 3,48 | 19,89 | -2,04559E-09 |
| PC20 | 17,33 | 4,92 | 3,61 | 19,22 | -2,0369E-09 |



**Monte Carlo**

|  |
| --- |
| Nº Pontos Gerados |
| 1100000000 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Nº Núcleos | | Nº Núcleos | |  |
|  | 4 | 12 | 4 | 12 |  |
| NºThreads | Tempo(s) | Tempo(s) | F | F | Erro (pi - π) / π \* 100 |
| Sequencial | 60,55 | 56,17 | - | - | 2,60E-03 |
| PC1 | 59,43 | 41,42 | - | - | -1,18E-03 |
| PC2 | 32,47 | 20,82 | 2,73 | 4,40 | 5,62E-04 |
| PC4 | 18,48 | 10,45 | 4,04 | 6,83 | 3,88E-04 |
| PC8 | 19,14 | 5,87 | 3,47 | 10,80 | -1,44E-03 |
| PC10 | 19,33 | 4,81 | 3,37 | 12,86 | -2,78E-04 |
| PC20 | 20,61 | 15,74 | 3,04 | 3,70 | 1,65E-04 |



A variável “F” corresponde ao valor calculado a partir da lei de amdahl, na qual foram utilizados os respetivos tempos das versões sequenciais e a respetiva versão concorrente, o resultado para os melhores tempos está realçado a verde na tabela considerado os resultados do portátil de 4 núcleos.

Podemos tirar como conclusão que no caso do método de Monte-Carlo tal como seria de esperar o número de threads que obteve melhor resultado foi o PC4 ou seja 4 threads pois o computador a ser executado também tinha 4 núcleos e o rácio de numero de threads e quantidade de núcleos deve ser 1:1,

Algo que não se sucedeu com o método de Séries de Gregory-Leibniz, não temos a certeza do porque de isto se ter sucedido, tendo sido neste a versão mais rápida a que tinha 20 threads sendo que nesta situação existe um rácio de 5 vezes mais threads do que núcleos no sistema, o que deveria implicar uma diminuição no tempo de execução medido tal como verificado no método de Monte-Carlo.

3. Conclusões

Com este trabalho conseguimos aplicar os conhecimentos adquiridos na unidade curricular Programação Concorrente e Distribuída, é com muito prazer do grupo que conseguimos concluir todas as etapas apresentadas pelo docente neste trabalho, foi com pena nossa de que alguns dos resultados obtidos não tenham sido de acordo com o esperado.