Raphael Ribeiro Faria RA:104120
João Victor de Mesquita RA:102028

Professor: Dr. Álvaro Luiz Fazenda
Disciplina: Programação Concorrente e Distribuída


# Atividade 1 - Calculando distância euclidiana com coordenadas polares usando Threads

## Problema 1:

Foi proposto que fosse implementado dois programas concorrentes para calcular a distância euclidiana de pares de pontos em coordenadas polares, também conhecido por distância radial , dispostos em vetores, sendo um programa na linguagem C utilizando *pthreads*, e outra na linguagem Java utilizando *JAVAthreads*. Os vetores são alocados dinamicamente, do tipo *double,* com tamanho *N,* considerado como $10^5$ e $10^7$, e o número de *threads* variando da seguinte forma: 1, 2, 4 e 8. A fórmula para o cálculo da distância euclidiana é dado pela fórmula abaixo:

Distância Euclidiana = $\sqrt{r_1^2 + r_2^2 - 2r_1 r_2 \cos(\theta_1 - \theta_2)}.$

Os vetores *r1, r2, Θ1 e Θ2* (correspondente aos pares: (r1,Θ1) e (r2,Θ2)) são preenchidos com valores aleatórios através da função *rand()*, variando de 0 a 100 os valores para as coordenadas radiais e 0 a 360 para os valores de coordenadas angulares.

O resultado esperado do cálculo corresponde à um vetor *D,* com as distâncias calculadas para todos os pares de pontos existentes.

Para a realização dos testes feitos, foi usado o computador Dell Inspiron com processador Intel Core i5-5200U, que possui dois núcleos físicos com Hyper-Threading de velocidade 2,2 - 2,7 GHz (2 Núcleos: 2,5 GHz), além de 1 TB e 8GB de memória principal e memória RAM, respectivamente. Os algoritmos foram executados utilizando o compilador da IDE Code Blocks MinGW GNU Compiler de versão 32 bits, sendo o sistema operacional da máquina o Windows 10 na versão de 64 bits.

Para o cálculo de todas as distâncias desejadas, devido ao elevado valor de N, foram divididos os cálculos entre as threads, realizando a tarefa de forma concorrente e obtendo um vetor de resultados ao final do experimento, como já mencionado antes.

- **Linguagem C**

As tabelas abaixo ilustra o tempo calculado para a execução total do problema com a utilização da função *gettimeofday()*, utilizado também para o cálculo apenas das distâncias euclidianas, com os devidos valores de N e número de threads usado.

| Tempo total | | |
|---|---|---|
| | Tamanho de N | |
| Threads | 10^5 | 10^7 |
| 1 | 95 ms | 3585 ms |
| 2 | 61 ms | 2116 ms |
| 4 | 45 ms | 1612 ms |
| 8 | 43 ms | 1599 ms |

| Tempo do cálculo da distância Euclidiana | | |
|---|---|---|
| | Tamanho de N | |
| Threads | 10^5 | 10^7 |
| 1 | 71 ms | 2884 ms |
| 2 | 34 ms | 1137 ms |
| 4 | 22 ms | 735 ms |
| 8 | 12 ms | 584 ms |

Como mencionado acima, podemos perceber que até um número de 4 threads é possível obter um desempenho significativo do algoritmo, devido à capacidade de Hyper-Threading da máquina, porém, com 8 threads já não se obtém um desempenho esperado, já que o computador não possui a capacidade de executar 8 tarefas em paralelo.

- **Linguagem Java**

Analogamente, foram realizados os mesmo experimentos e verificados os mesmos resultados na linguagem Java, como ilustrado abaixo:

| Tempo total | | |
|---|---|---|
| | Tamanho de N | |
| Threads | 10^5 | 10^7 |
| 1 | 140 ms | 4089 ms |
| 2 | 80 ms | 3187 ms |
| 4 | 50 ms | 1852 ms |
| 8 | 48 ms | 1562 ms |

| Tempo do cálculo da distância Euclidiana | | |
|---|---|---|
| | Tamanho de N | |
| Threads | 10^5 | 10^7 |
| 1 | 102 ms | 3567 ms |
| 2 | 69 ms | 2598 ms |
| 4 | 35 ms | 1145 ms |
| 8 | 28 ms | 1081 ms |

## Problema 2:

Para a segunda parte do problema, foi solicitado que o mesmo código seja utilizado, porém agora com a funcionalidade extra de encontrar também a maior distância entre as calculadas.

- **Linguagem C**

Para realizar essa tarefa no algoritmo desenvolvido em C, foi criado um vetor de tamanho MAX_THREADS onde cada thread armazena a maior distância calculada entre os seus pontos no espaço destinado. Por exemplo: A thread 1 armazena o maior valor calculado por ela em vetor[1], a thread 2 armazena o maior valor calculado por ela em vetor[2] e consecutivamente. Ao final do programa, é feita apenas uma varredura entre os valores enviados pelas threads e encontrado o maior entre eles.

Essa varredura entre os valores é feita de maneira muito rápida, porém, a função *gettimeofday()* foi colocada no código de forma que o tempo calculado seja a espera do término das threads e a varredura em si, como mostrada nas tabelas abaixo:

| Tempo total | | |
|---|---|---|
| | Tamanho de N | |
| Threads | 10^5 | 10^7 |
| 1 | 51 ms | 3060 ms |
| 2 | 31 ms | 2159 ms |
| 4 | 15 ms | 1542 ms |
| 8 | 37 ms | 1700 ms |

| Tempo do cálculo da maior distância | | |
|---|---|---|
| | Tamanho de N | |
| Threads | 10^5 | 10^7 |
| 1 | 48 ms | 2134 ms |
| 2 | 16 ms | 1192 ms |
| 4 | 6 ms | 728 ms |
| 8 | 4 ms | 632 ms |

Assim como no problema 1, foi possível perceber que a partir do uso de 8 threads a eficiência alcançada já não era tão satisfatória, sendo que no caso do tempo total acaba tendo uma eficiência até inferior ao processamento com 4 threads.

- **Linguagem Java**

Analogamente, foram realizados os mesmos testes e verificações no algoritmo em Java, cujos resultados são representados abaixo:

| Tempo total | | |
|---|---|---|
| | Tamanho de N | |
| Threads | 10^5 | 10^7 |
| 1 | 122 ms | 3549 ms |
| 2 | 85 ms | 2653 ms |
| 4 | 52 ms | 1872 ms |
| 8 | 41 ms | 1753 ms |

| Tempo do cálculo da maior distância | | |
|---|---|---|
| | Tamanho de N | |
| Threads | 10^5 | 10^7 |
| 1 | 81 ms | 2672 ms |
| 2 | 64 ms | 1849 ms |
| 4 | 38 ms | 1395 ms |
| 8 | 28 ms | 1056 ms |