Projeto 2 – Inversão de matriz com threads

Sistemas Operacionais TT304 A

Grupo Equipe Rocket

Alice Mantovani 193539

Guilherme Masao 217250

Limeira – SP

Junho/2019

Objetivo e Especificações:

O trabalho deveria utilizar multithreads para rotacionar uma matriz N x M, sendo N número de linhas e M de colunas, o programa deveria ser escrito para o sistema operacional Linux e obrigatoriamente utilizar a biblioteca de extensão POSIX Threads.

Os números da matriz podem ser reais ou não.

Os dados da matriz original devem vir de um arquivo e a matriz resultante deve ser gravada em arquivo com a extensão .rot.

O programa deve ser testado para 2, 4, 8 e 16 threads, com matrizes 1000 x 1000.

Solução do problema:

Com a finalidade de solucionar o problema foi desenvolvido um código em C que rotaciona uma matriz que vem de um arquivo de entrada e a imprime no arquivo de saída.

Para esse caso, é utilizado o conceito de threads que, dividem a função de rotacionar a matriz em partes, e cada uma fica responsável por um dado, que respeita determinada função matemática (explicada posteriormente no algoritmo em alto nível), denominado método do “Canguru” visto que ele salta uma determinada quantidade de posições e pega o dado que deseja inverter.

Algoritmo de alto nível:

1. Foram declaradas as bibliotecas que deveriam ser usadas no código, no caso, “stdio.h”, “stdlib.h”, “pthread.h”, “time.h”.
2. Declara-se a struct a ser usada com os seguintes membros: ponteiro de ponteiro de “matriz”, “num_c”, “num_l”, “posicao_atual”. “t” sendo que representam, respectivamente, a matriz, o número de colunas, o número de linhas, a posição atual que a thread deve pegar e o número de threads.
3. Cria-se o protótipo da função de rotacionar a matriz.
4. Há a declaração da variável global que é o ponteiro de ponteiro de matriz inversa que é utilizado em ambas funções do código.
5. Inicializa-se a main com dois parâmetros que permitem a execução do comando atoi.
6. Inicializa-se uma struct que será utilizada durante o código, sendo essa do tipo da primeira.
7. Inicializa-se as variáveis num_l, num_c, num_t (variáveis locais), o arquivo de entrada e o de saída com os valores/endereços passados pelo usuário no terminal.
8. Verifica-se se os arquivos de entrada e saída são abertos ou não.
9. Aloca-se dinamicamente duas matrizes, uma nova que é ponteiro de ponteiro e a outra inicializada como variável global, a fim de não utilizar uma matriz estática que ocupa mais espaço e consequentemente gasta mais recurso.
10. Posteriormente, lê-se o arquivo de entrada a partir da função fscanf.
11. A partir da biblioteca “time.h” inicializa-se duas variáveis que contam o início da execução e o fim em determinado período do código.
12. Inicia-se a variável que conta o tempo inicial do processamento com threads.
13. Cria-se um for em que a condição é que i, que se inicializa com o valor de 0 seja menor que o número de threads desejadas.
14. Com isso, entra-se no for e há uma atribuição de valores a um vetor de struct’s pois se não for um vetor há uma incompatibilidade na hora de atribuição de valores que gera, posteriormente, um erro de segmentação.
15. Confere-se o número de threads desejadas e entra no seu respectivo caso
16. Nesse momento, criam-se as threads com o auxílio da biblioteca <pthread.h> e chama-se a função de rotacionar.
17. Em caso, de número de threads inválido, o código exibe uma mensagem de erro.
18. Entra-se na função de inverter a matriz e inicializa-se variáveis auxiliares.
19. Há a atribuição da posição atual em que a thread se encontra na matriz, em caso de não ter essa atribuição ocorre um problema de segmentação por conta de um somatório posterior.
20. Cria-se um laço em que a condição é que: a posição em que a thread se encontra seja menor do que o número de linhas multiplicado pelo número de colunas da matriz da struct

21. Para encontrar o local da matriz em que a thread se encontra utiliza-se das fórmulas em que a linha_atual é igual a posição em que se encontra a thread dividido pelo número de colunas da matriz.
22. Já para descobrir em que coluna encontra-se a thread executa-se a fórmula: posição em que se encontra a thread mod número de colunas da matriz.
23. Verifica-se se a matriz é quadrada ou não
24. Caso não seja, a matriz invertida inverte-se coluna_atual e linha_atual do modelo normal de matriz
25. Além disso, na linha da matriz não rotacionada executa-se a fórmula: número de linhas subtraído pela a linha da matriz da posição atual da thread - 1
26. Já, na coluna utiliza-se o número da coluna da posição atual da thread.
27. No caso da matriz quadrada, nos locais onde se utilizava linha substitui-se por colunas.
28. Por fim, soma-se em posição inicial da thread o número de threads para que possa ser executado o método “canguru” em que há a separação da matriz em que cada thread pega uma posição e a próxima em que trabalha é sua posição mais o seu número.
29. Saindo da função que executa as threads, tem a função join que permite que uma thread só seja executada após o término da outra, essa tarefa também resolve problemas de segmentação que possam ocorrer.
30. Pega-se o tempo final da execução das threads.
31. Subtrai-se o tempo final pelo inicial e multiplica-se 1000, posteriormente, divide essa conta por CLOCK_PER_SEC
32. Imprime o resultado na tela, considerando que esse tempo apresenta-se em ms.
33. Passa-se a matriz invertida para o arquivo de saída.
34. Fecha-se os arquivos de entrada e saída.
35. Encerra-se o programa.

Configurações do computador:

As configurações do computador em que os testes foram realizados são:

- Processador: Intel(R) Core(™) i7-770K CPU @ 4.20 GHz 4.20 GHz

- Memória instalada (RAM) : 16,0 GB

- Tipo do Sistema: Sistema operacional de 64 bits

Instruções para execução:

Quando executado durante os testes o código foi compilado da seguinte forma:

gcc nome_do_programaC -o nome_do_executável -pthread

E foi executado da seguinte forma:

./nome_do_executável n_linhas n_colunas n_threads ArquivoDeEntrada ArquivoDeSaida

Gráfico:

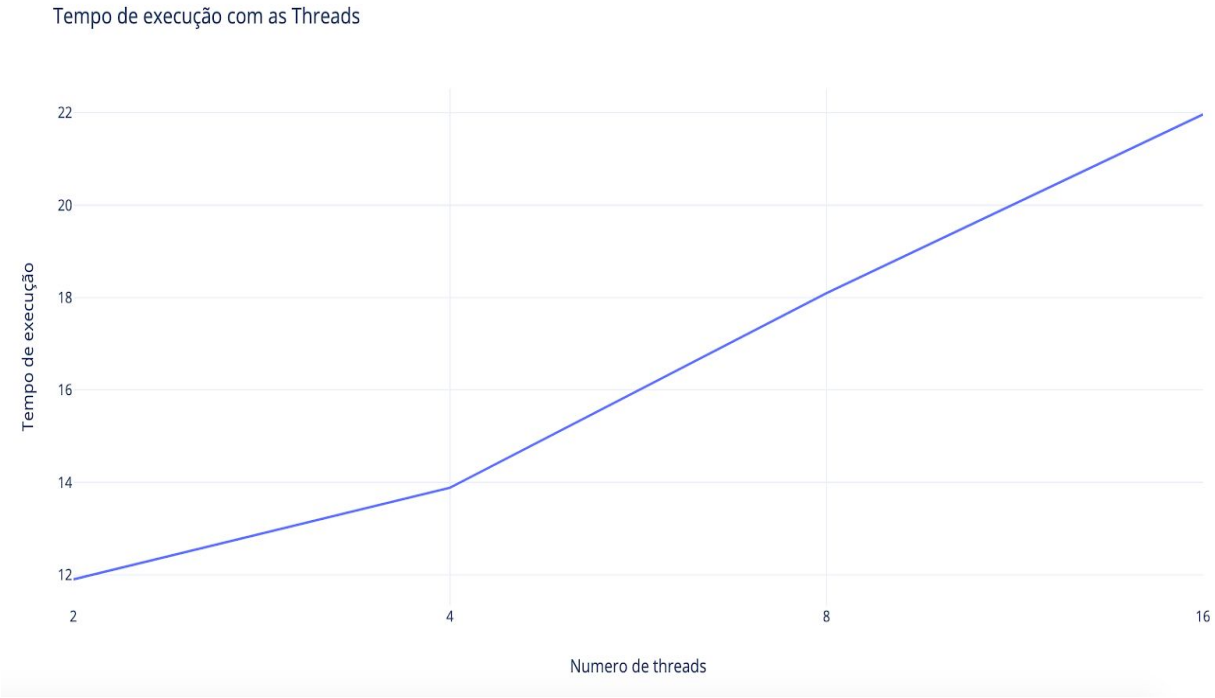

Imagem 1 - Gráfico (Tempo de execução x Número de threads)

O gráfico utiliza-se dos tempos de execução calculados durante o período de testes do código. No caso dessa tabela os Tempos de execução são:
Para 2 threads - 11,902 ms
Para 4 threads - 13,885 ms
Para 8 threads - 18,096001ms
Para 16 threads - 21,964001ms

Conclusão:

Analisando o gráfico gerado e, a partir do comportamento do método adotado para a execução das threads (“canguru”) cremos que quanto maior o número de threads por necessitar que essas threads façam muitos cálculos e façam muitos pulos na matriz, por conta de cada thread pegar apenas um dado, em vez de pegar uma porção da matriz como é feito em outros métodos. Com isso, faz sentido o fato de que quanto maior o número de threads maior o número de tempo de execução, lembrando que esse tempo de execução ainda tem uma leve oscilação.

Outro ponto que notamos é que mesmo sendo um arquivo muito grande (matriz [1000][1000]), o processamento com as threads fez com que o tempo de execução tenha sido relativamente baixo, esperávamos um número maior no tempo de execução que foi desmentido durante o processo de treinamento.

Link do Github:

https://github.com/GuiMasao/SistemasOperacionais

O código final chama-se “código_final.c”

Link para o Google Drive (vídeo):

https://drive.google.com/file/d/15tQhfmL9IR2emYBlnltEd6W9VF3cXMnX/view?usp=sharing