Universidade Estadual de Campinas
Faculdade de Tecnologia

Sistemas Operacionais - TT304

Projeto - Rotacionar matriz em 90° usando múltiplos Threads

Prof. André Leon S. Gradvohl, Dr.

Grupo : Clube das Winx

Diovan Queiroz	RA: 169975
Bruno Rossetto	RA: 214066
Luiz Augusto Duarte	RA: 182998

Limeira-SP
2019

ÍNDICE

1. Introdução

1.1 Propósito

O projeto busca implementar um código que seja capaz de rotacionar em 90° uma matriz quadrática utilizando N threads, onde N é definido pelo usuário em sua compilação no sistema operacional Linux ou similares.

1.2 Visão Geral

Os dados para o preenchimento da matriz que sofrerá rotação estão separados em um arquivo com o formato .dat criado previamente. Com isso, o código desenvolvido terá que ler os dados, realizar a conversão em 90° com a implementação de múltiplos threads dessa matriz e então salvá-la em outro arquivo com o formato .rot.

A sintaxe para a execução do programa no terminal do sistema operacional deverá ser: N M T <arquivo_de_entrada.dat> <arquivo_de_saida.rot> - onde N = número de linhas, M = número de colunas e T = número de threads. Após a execução, o arquivo de saída estará com o mesmo nome do arquivo de entrada, mudando somente o formato.

2. Desenvolvimento

2.1 Código Fonte

Segue abaixo o código fonte implementado pelo grupo:

```
#include <stdio.h>
#include <stdlib.h>
#include<time.h>
#include<pthread.h>

//struct que será passada como parâmetro para a função que gira a matriz 90 graus para a direita
typedef struct dados {
   int l;
   int c;
   int t;
   int posicao;
```

```
    double **matriz;
        double **matrizInvertida;
}DADOS;

//função que retorna a linha da posição atual em que a thread está
int retornaLinha(int posicaoAtual, int coluna){
    return posicaoAtual/coluna;
}

//função que retorna a coluna da posição atual em que a thread está
int retornaColuna(int posicaoAtual, int coluna){
    return posicaoAtual%coluna;
}

//Função que gira a matriz 90 graus para a direita, as threads percorrem a matriz de forma intercalada
void *inverte(void *argts){
    DADOS *argumentos = argts;

    int posicaoatual = argumentos->posicao;

    //Percorre a matriz da posição atual até o final dela.
    while (posicaoatual<(argumentos->l * argumentos->c)){
        if(argumentos->l != argumentos->c){ //se a matriz não for quadrada
            int linhaAtual = retornaLinha(posicaoatual, argumentos->c);
            int colunaAtual = retornaColuna(posicaoatual, argumentos->c);
                argumentos->matrizInvertida[colunaAtual][linhaAtual] = argumentos->matriz[argumentos->l - linhaAtual - 1][colunaAtual];
            posicaoatual = posicaoatual + argumentos->t;
        }else{ //se a matriz for quadrada
            int linhaAtual = retornaLinha(posicaoatual, argumentos->c);
            int colunaAtual = retornaColuna(posicaoatual, argumentos->c);
                argumentos->matrizInvertida[linhaAtual][colunaAtual] = argumentos->matriz[argumentos->c - colunaAtual - 1][linhaAtual];
            posicaoatual = posicaoatual + argumentos->t;
        }

    }
}

int main(int argc, char *argv[])
{
    DADOS argumentos[16];
    int l = atoi(argv[1]); //numero de linhas
    int c = atoi(argv[2]); //numero de colunas
    int t = atoi(argv[3]); //numero de threads
```

```
        FILE *file = fopen(argv[4], "r");


  //alocação dinâmica da matriz
        double **matriz = (double **)malloc(l * sizeof(double *));
        for (int i = 0; i < l; i++){
                matriz[i] = (double*) malloc(c * sizeof(double));
                for (int j = 0; j < c; j++){
        fscanf(file, "%lf", &matriz[i][j]);
        }
 }
  //alocação dinâmica da matriz invertida
double **matrizInvertida = (double **)malloc(c * sizeof(double *));
        for (int i = 0; i < c; i++){
                matrizInvertida[i] = (double*) malloc(l * sizeof(double));
                for (int j = 0; j < l; j++){
        matrizInvertida[i][j] = 0;
        }
 }

        pthread_t threads[t];

        if (!file)
        printf ("Erro na abertura do arquivo. \n");

  float inicial = clock(); //Inicializa a contagem de tempo das threads

  //da valor as variaveis da struct e executa a função inverte com as threads.
  for(int i=0; i<t; i++){
        argumentos[i].matriz = matriz;
        argumentos[i].matrizInvertida = matrizInvertida;
        argumentos[i].l = l;
        argumentos[i].c = c;
        argumentos[i].t = t;
        argumentos[i].posicao = i;
        pthread_create(&threads[i], NULL, &inverte, (void *)&argumentos[i]);
  }
  for(int i=0; i<t; i++){
        pthread_join(threads[i], NULL);
  }

  float final = clock(); // Finaliza a contagem de tempo das threads


        FILE *fileSaida = fopen(argv[5], "w");
```

```
        //escreve a matriz resultado (matriz invertida) em outro arquivo.
        for ( int i=0; i<c; i++ ){
         for (int j=0; j<l; j++ )
         {
                fprintf(fileSaida, "%0.2lf ", matrizInvertida[i][j]);
         }
         fprintf(fileSaida, "\n");
        }

   fclose(file);
   fclose(fileSaida);

    float tep = (final - inicial) * 1000.0 / CLOCKS_PER_SEC; //Cálcula o tempo gasto com N threads
    printf("Tempo de execucação com [%d] threads: %f milissegundos \n", t,tep);     //Imprime o tempo de execução com N threads


        return 0;
}
```

Como pode-se observar, o código contém 4 funções, sendo elas:

- *retornaLinha* - Recebe como parâmetro a posição atual e a coluna e retorna a linha da posição atual em que a thread se encontra dividindo a posição atual pela coluna.
- *retornaColuna* - Recebe como parâmetro a posição atual e a coluna e retorna a coluna da posição atual em que a thread se encontra achando o resto da divisão da posição atual por coluna.
- *inverte* - Recebe uma struct como parâmetro, percorre a matriz da posição atual até o final dela, faz uma verificação se é uma matriz quadrática ou não e implementa o algoritmo de inversão de matriz em 90 graus. Após isso, é definido que a próxima posição da thread atual será a posição atual somado com o número de threads.
- *main* - Responsável pela abertura, leitura e fechamento dos arquivos (.dat) que contém os dados da matriz original e escrita da matriz invertida em outro arquivo (.rot), alocação dinâmica de memória para comportar as matrizes, inicialização das threads necessárias, dá valor para as variáveis da struct e faz com que as threads se alinhem para a execução da função inverte. Após isso, executamos a função *join* para recuperação dos dados que elas estavam gerando, para sincronização da rotação da matriz. Usamos a função *clock* para conseguirmos o tempo de processamento com N threads e fazer a análise e comparação entre eles, iniciando após a chamada do thread e finalizando após o *join*.

e também utilizamos uma struct que contém as variáveis necessárias para o funcionamento da função *inverte*, que são: número de linhas, número de colunas, número de threads, posição atual da thread, matriz original e matriz invertida.

2.2 Instruções para compilação

Para a compilação no sistema operacional Linux e similares, precisamos ter primeiramente o código fonte em formato (.c) no mesmo diretório do arquivo de entrada e então executar no terminal os seguintes comandos :

- *gcc <NomeDoArquivo.c> -lpthread -o <NomeDoExecutável>*
- *./<NomeDoExecutável> <Número de linhas> <Número de colunas> <Número de Threads> <Arquivo de entrada.dat> <Arquivo de saída.rot>*

2.2 Testes Realizados

Vídeo dos testes realizados.
https://www.youtube.com/watch?v=XSUwprjlkV4&feature=youtu.be

Fizemos os testes conforme as imagens a seguir demonstram:

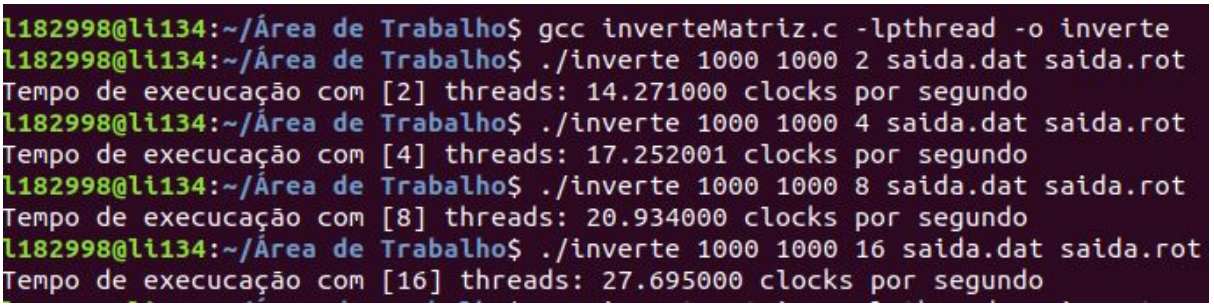

Figura 1 - Teste de execução em computador 1

Figura 2 - Teste de execução em computador 2

3. Resultados e Conclusão

3.1 Resultados

Com os testes realizados, obtivemos resultados dos 2 computadores que nos comprometemos a executar o código. Segue abaixo o gráfico de Tempo (milissegundos) X Threads que obtivemos com o computador 1.

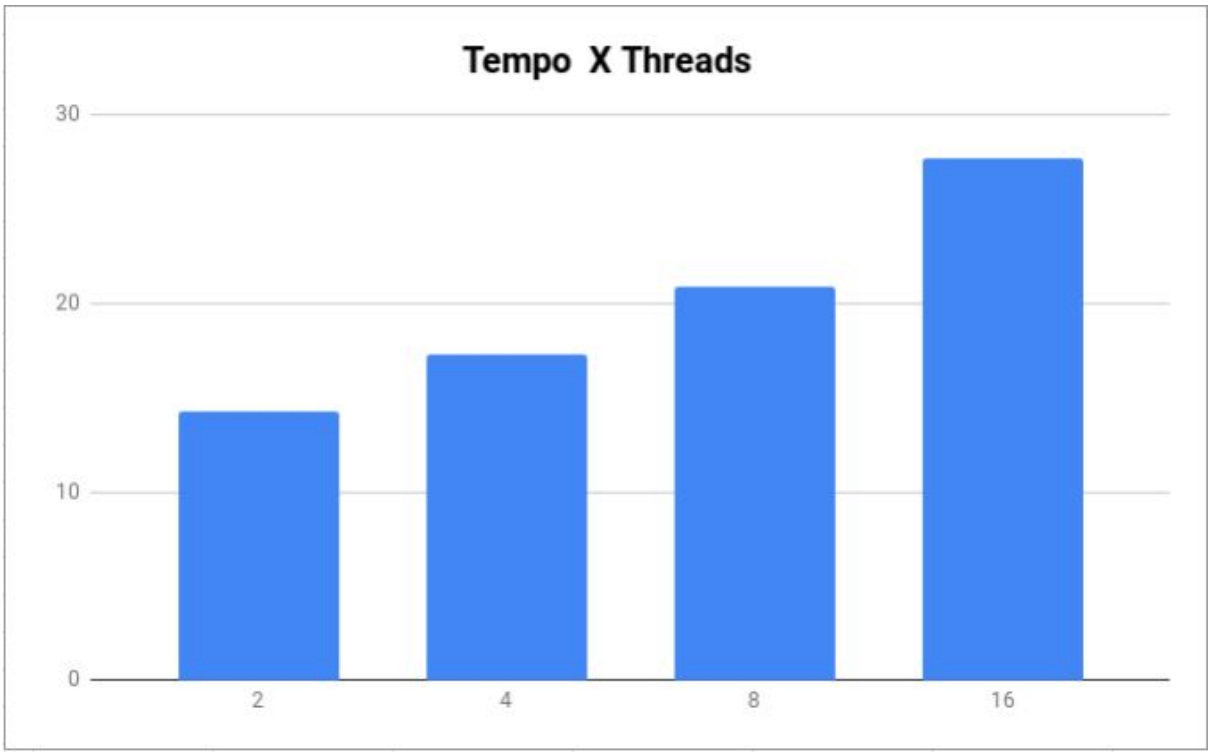

Gráfico 1 - Tempo X Threads em computador 1

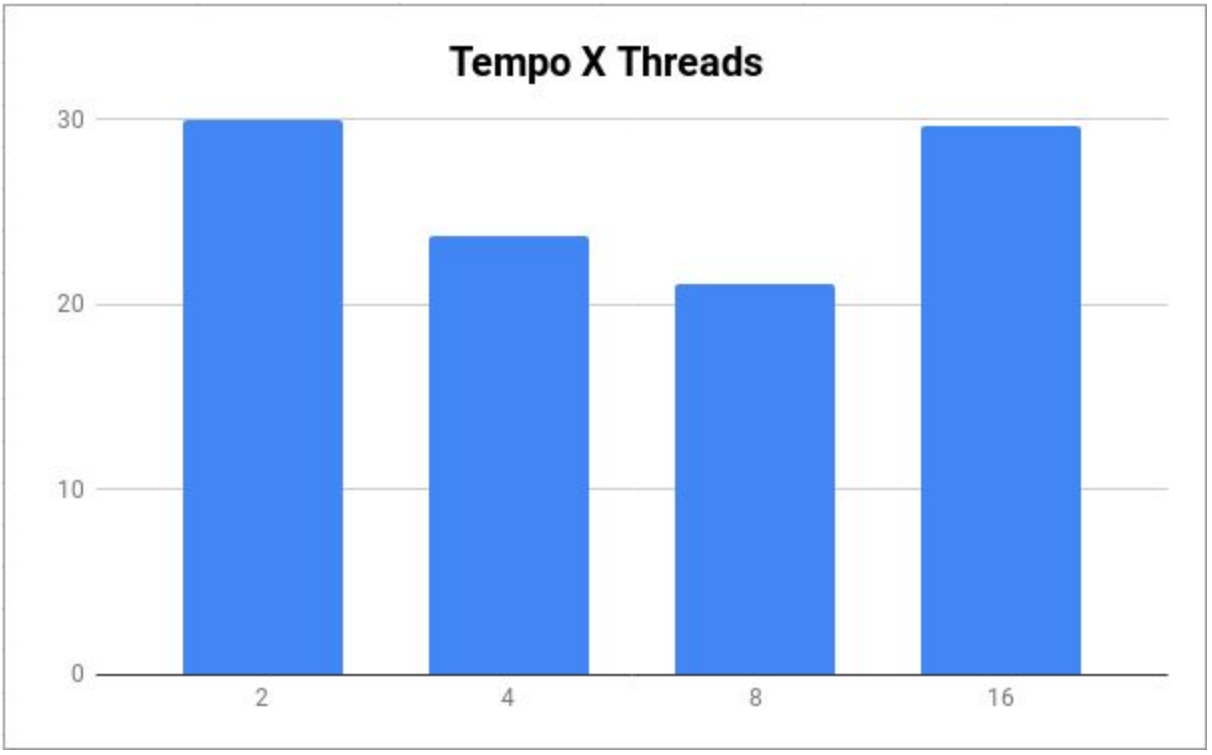

Gráfico 2 - Tempo X Threads em computador 2

Em ambos os gráficos, tempos no eixo vertical o tempo necessário para a execução das threads no programa em milissegundos, já no eixo horizontal a quantidade de threads utilizada para o teste, definida sempre na execução, pelo usuário. Como podemos observar

no *Gráfico 1* que foi testado no *computador 1*, quanto maior o número de threads maior o tempo de execução da inversão da matriz. Por outro lado, no *Gráfico 2* temos uma variação maior do tempo de execução em *computador 2*, com a mais rápida sendo com 8 threads e a mais lenta com 2 threads.

3.2 Conclusão

A partir dos resultados obtidos do tempo de inversão de matrizes NxM usando múltiplos threads, concluímos que o tempo de criação das threads não compensa o trabalho que ela tem, dependendo do sistema operacional e da quantidade de núcleos virtuais e reais presentes no processador de cada máquina.

Como já dito antes, foi utilizado 2 computadores para os testes de tempo de trabalho dos threads, o primeiro, denominado *computador 1*, tinha as seguintes configurações em seu processador:

```
l182998@li134:~/Área de Trabalho/geraMatrizAleatoria$ lscpu
Arquitetura:                    x86_64
Modo(s) operacional da CPU:     32-bit, 64-bit
Ordem dos bytes:                Little Endian
CPU(s):                         8
Lista de CPU(s) on-line:        0-7
Thread(s) per núcleo:           2
Núcleo(s) por soquete:          4
Soquete(s):                     1
Nó(s) de NUMA:                  1
ID de fornecedor:               GenuineIntel
Família da CPU:                 6
Modelo:                         158
Nome do modelo:                 Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz
Step:                           9
CPU MHz:                        800.020
CPU MHz máx.:                   4500,0000
CPU MHz mín.:                   800,0000
BogoMIPS:                       8400.00
Virtualização:                  VT-x
cache de L1d:                   32K
cache de L1i:                   32K
cache de L2:                    256K
cache de L3:                    8192K
CPU(s) de nó0 NUMA:             0-7
```

Figura 3 - Detalhes do computador 1

Portanto, nota-se que o *computador 1* tem 4 núcleos reais e cada um deles contém 2 threads, o que pode ser observado no *Gráfico 1* que apresenta mais agilidade na execução do programa. A quantidade de tempo cresce conforme a quantidade de threads aumenta, porque quanto mais threads mais “demora” temos, afinal, 1 thread para recuperar seus dados precisa esperar outros threads acabarem suas tarefas. Além disso, nota-se que há um crescimento uniforme de 2, 4 e 8 threads, pois o processador em questão contém somente 8 threads, mas quando é solicitado fazer a execução com 16 threads ele tem o

tempo de criação de 8 threads virtuais embutidos, aumentando de forma significativa o tempo de milissegundos.

Já o *computador 2* contém as seguintes configurações em seu processador:

```
l182998@li020:~/Área de Trabalho/geraMatrizAleatoria$ lscpu
Arquitetura:                    x86_64
Modo(s) operacional da CPU:     32-bit, 64-bit
Ordem dos bytes:                Little Endian
CPU(s):                         4
Lista de CPU(s) on-line:        0-3
Thread(s) per núcleo:           1
Núcleo(s) por soquete:          4
Soquete(s):                     1
Nó(s) de NUMA:                  1
ID de fornecedor:               GenuineIntel
Família da CPU:                 6
Modelo:                         58
Nome do modelo:                 Intel(R) Core(TM) i5-3330 CPU @ 3.00GHz
Step:                           9
CPU MHz:                        1596.440
CPU MHz máx.:                   3200,0000
CPU MHz mín.:                   1600,0000
BogoMIPS:                       5986.42
Virtualização:                  VT-x
cache de L1d:                   32K
cache de L1i:                   32K
cache de L2:                    256K
cache de L3:                    6144K
CPU(s) de nó0 NUMA:             0-3
```

Figura 4 - Detalhes do computador 2

Como apresentado na *Figura 4*, o *computador 2* contém 4 núcleos reais com threads únicos em cada um. Neste caso, observando o *Gráfico 2*, podemos inferir que os threads com números centrais, que são: 8 e 4, conseguiram os melhores desempenhos, respectivamente. O motivo disso é que com base na estrutura física quando se executa com apenas 2 threads, elas acabam sendo sobrecarregadas e demorando muito tempo para executar as tarefas, igualmente ao desempenho com 16 threads, porém, com 16 threads é necessário criar 12 threads virtuais para a execução ser realizada como o usuário desejou.

Por fim, podemos concluir que quando se trata de computadores semelhantes as configurações de processador do *computador 1* temos melhores desempenhos com menos threads, já com computadores parecidos com o processador do *computador 2* há um balanço que faz com que números medianos de threads sejam mais eficientes.

3.3 Endereço do Repositório Git

https://github.com/brunorp/InverteMatriz.git