FT UNICAMP, Junho de 2018

# Localiza na matriz

Relatório do projeto (https://github.com/felipemarchi/sistemas-operacionais)

## Descrição do problema

O projeto solicitou: (1) criar um programa que utilize múltiplas threads para buscar um valor determinado pelo usuário em uma matriz e (2) analisar o desempenho desse programa utilizando 02, 04, 08 e 16 threads.

Junto à descrição do problema encontrou-se restrições e obrigações, das quais retirou-se os seguintes critérios:

- O programa deve ser compilado em sistema operacional Linux;
- POSIX Threads é uma biblioteca obrigatória do programa;
- A matriz contém números reais, positivos ou negativos, aleatórios e não ordenados;
- A matriz pode ter valores repetidos.
- O elemento buscado pode não estar contido da matriz;
- As entradas do programa são: número de linhas da matriz (M), número de colunas da matriz (N), nome do arquivo da matriz, número de threads a serem utilizadas, e o valor a ser buscado na matriz;
- As saídas do programa são: as posições onde está cada elemento encontrado, ordenadas de acordo com a posição na matriz. Ou, se for o caso, uma mensagem informando que o elemento não foi encontrado.
- Os testes devem considerar valores maiores ou iguais a 100 para M e N.

## Solução do problema

Primeiramente, com base nos critérios retirados da descrição do problema, o grupo optou por programar em linguagem C e utilizar 03 das 05 entradas do usuário para gerar uma matriz aleatória a cada execução do programa.

Assim, foi desenvolvido um esboço da rotina do programa - ilustrado pelo fluxograma da Figura 1 e descrito logo abaixo.

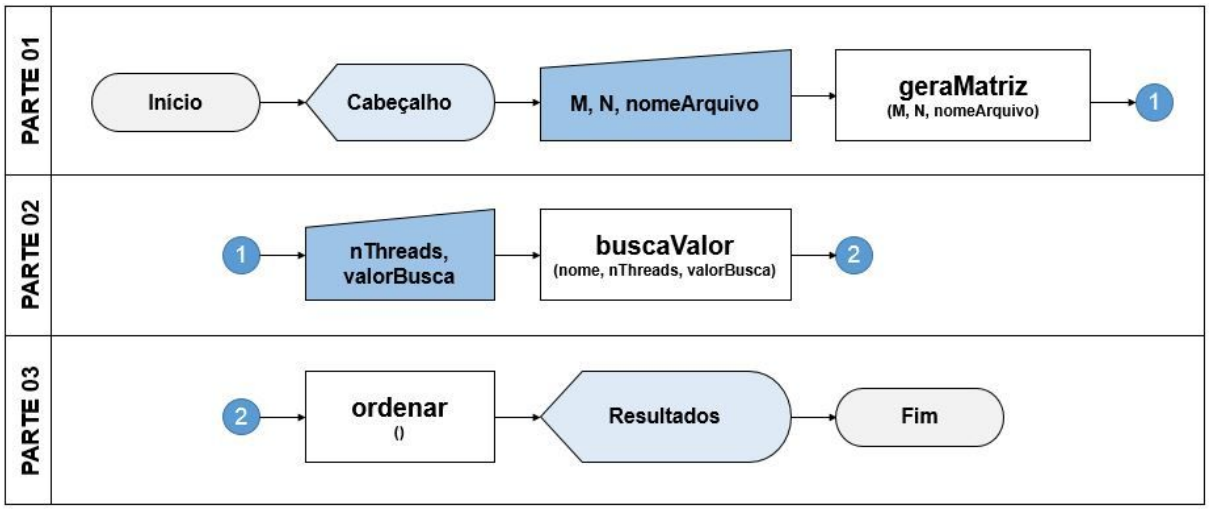


Figura 1 - Fluxograma da rotina do programa.

Descrição da rotina do programa:

- **PARTE 01 (GERAR MATRIZ)**

Nesta sessão o usuário deverá fornecer os 03 primeiros valores de entrada para gerar a matriz: número de linhas da matriz (M), número de colunas da matriz (N), e o nome do arquivo da matriz.

A função **geraMatriz()** deverá chamar a execução de um programa auxiliar desenvolvido pelo Professor Dr. André Leon Sampaio Gradvohl que, por sua vez, criará um arquivo com a matriz de acordo com os parâmetros passados.

- **PARTE 02 (BUSCAR VALOR)**

Para esta sessão o usuário deverá entrar com os valores restantes: número de threads a serem utilizadas, e o valor a ser buscado na matriz.

A função **buscaValores()** deverá criar as threads passando os argumentos necessários para busca, além de impedir que a função main() termine antes do retorno das threads.

As threads em si são uma função genérica que deverá percorrer o arquivo em busca do valor solicitado pelo usuário a partir de um ponto da matriz, calculado para cada thread individualmente.

- **PARTE 03 (EXIBIR RESULTADOS)**

Esta última parte será responsável pela saída do programa e não solicita mais nada do usuário. Se o valor buscado estava presente na matriz, então as posições encontradas deverão ser listadas ordenadamente, caso contrário, uma mensagem deverá informar que o elemento não foi encontrado.

Toda posição encontrada durante a execução das threads será salva em uma lista encadeada e a função **imprimeResultados()** simplesmente fará a verificação se esta lista encontra-se vazia, para exibir a mensagem, ou as posições ordenadas.

Para desenvolver o que planejou-se acima, as tarefas foram divididas entre o grupo de acordo com a tabela:

| Responsável | Tarefa | Detalhes | Data limite |
|---|---|---|---|
| Pedro | Estrutura de pastas do repositório | - Copiar “testeMatrizes” e removê-la do projeto.<br>- Criar pasta “arquivos” com os três modelos (matriz, relatório, vídeo). | 17/05 |
| Pedro | Entrada do usuário | Tratar as 05 entradas do usuário e gerar a matriz utilizando o programa auxiliar. | 20/05 |
| Felipe | Criar threads | divisão = m.n / t<br>For (i < t) criar threads (i, divisão, etc) | 23/05 |
| Brenda | Algoritmo de busca | Tratar os argumentos para realizar a busca.<br>Se sucesso, salvar posição dinamicamente. | 26/05 |
| Brenda | Algoritmo de ordenação | Se existir, ordenar posições e printá-las.<br>Ou então “Elemento não encontrado” | 29/05 |
| Felipe | Documentar | // Comentar sobre o código | 01/06 |
| **Todos** | Teste | Testar todo o programa. Não commitar. | 04/06 |
| Brenda | Relatório (2) | Solução do problema / Conclusão | 07/06 |
| Felipe | Make/Relat. | Arquivo makefile / Instruções para compilar. | 07/06 |
| Pedro | Vídeo/Relat. | Arquivo de vídeo / Tempos de execução. | 10/06 |
| **Todos** | Inspeção | Conferir arquivos. Enviar Moodle e GitHub. | 14/06 |

## Instruções para compilar/executar

A pasta *projeto* contém os arquivos fonte e um *makefile* para o programa. Mova-se até essa pasta via terminal Linux e insira o comando “make” (sem aspas) para executar o arquivo *makefile* que contém as instruções para compilar e gerar o programa automaticamente.

**$ make**

Após compilado, execute o programa através do comando “./Projeto1” (sem aspas) para analisar o resultado do projeto.

**$ ./Projeto1**

Como o trabalho faz ligação ao tempo de execução do processo, teste o comando “time ./Projeto1” (sem aspas) para também fazer esta análise. Após finalizado, o sistema retornará três tempos de execução, que são, respectivamente, o tempo real entre a invocação e a finalização do comando, o tempo de CPU empregado durante a execução e o tempo de sistema.

**$ time ./Projeto1**

# Resultados

Foram reunidos os tempos de execução do programa para 02, 04, 08 e 16 threads em duas matrizes distintas, tal que a primeira possui 100 linhas x 100 colunas (10.000 elementos) e a segunda 1000 linhas x 1000 colunas (1.000.000 de elementos).

| 100 Linhas x 100 Colunas | | |
|---|---|---|
| Número de Threads | Tempo real em ms | Tempo de CPU (user) em ms |
| 2 | 17 | 8 |
| 4 | 20 | 12 |
| 8 | 24 | 12 |
| 16 | 31 | 20 |

Figura 2 - Tempos de execução do programa com matriz de 10.000 elementos.

| 1000 Linhas x 1000 Colunas | | |
|---|---|---|
| Número de Threads | Tempo real em ms | Tempo de CPU (user) em ms |
| 2 | 1264 | 1376 |
| 4 | 1406 | 1628 |
| 8 | 1632 | 2220 |
| 16 | 2242 | 3364 |

Figura 3 - Tempos de execução do programa com matriz de 1.000.000 de elementos.

Veja a seguir a representação gráfica das tabelas acima, os tempos de execução (real e de CPU) do programa para 2, 4, 8 e 16 threads nas matrizes de 10.000 e 1.000.000 de elementos.

Pode-se notar em ambos os gráficos que a ascensão da quantidade de threads resulta em perda de desempenho. Certamente isso ocorre por dois motivos: a criação de novas threads leva tempo e o problema não tem uma boa solução ou é simples demais para a utilização de muitas threads.


Figura 4 - Gráfico dos tempos de execução da matriz de 10.000 elementos para 2, 4, 8 e 16 threads.


Figura 5 - Gráfico dos tempos de execução da matriz de 1.000.000 elementos para 2, 4, 8 e 16 threads.

# Conclusão

Apesar de não obter-se melhores tempos de execução conforme o número de threads aumentava, o grupo foi capaz de obter resultados satisfatórios, uma vez que conseguiu, sem muita dificuldade, aplicar no projeto o conceito de múltiplas threads, ou seja: as comparações realizadas durante a busca pelo elemento desejado foram divididas em T números de threads e realizadas de forma simultânea.

O código final da solução teve pequenas adaptações quanto ao seu planejamento, afinal este foi um grande facilitador para o desenvolvimento do projeto, assim como a plataforma GitHub.