Victor Gomes de Menezes

MULTIPLICAÇÃO DE MATRIZES QUADRADAS UTILIZANDO PROGRAMAÇÃO PARALELA COM PTHREADS

Trabalho apresentado a matéria de Tópicos Especiais em Internet das Coisas do Instituto Metrópole Digital, na Universidade Federal do Rio Grande do Norte, com fim de obtenção de nota na 2° (segunda) unidade do semestre letivo de 2019.1.

Universidade Federal do Rio Grande do Norte – UFRN

Curso de Bacharelado em Tecnologia da Informação

Natal-RN

Maio de 2019

Sumário

1 Introdução

Durante o ensino médio, quando se é introduzido o conceito de matrizes ao aluno, possivelmente será uma tarefa difícil enxegar as possibilidades e aplicações reais desta área da matemática. Sabe-se que as matrizes estão presentes em diferentes âmbitos da sociedade, seja na engenharia ou na planilha de gastos mensais. E uma das aplicações mais importantes do último século se encontra na área da tecnologia/computação, como por exemplo, em um dos adventos tecnológicos mais consagrados do século XX: o aparelho de reprodução de imagens e áudio instantâneos, ou simplesmente televisão.

A reprodução do que hoje é conhecido como vídeo nada mais é do que uma sequência de imagens em alta velocidade. Atualmente possuímos monitores que reproduzem vídeos a uma frequência de 144 fps (*frames* por segundo), mas estes valores podem chegar a milhares quando o intuito é a produção de vídeos em *slow-motion*. Quanto maior a taxa de imagens por segundo, maior o processamento de dados envolvido para que isto ocorra.

Imagens resumidamente são uma matriz de pixels. Cada pixel, no padrão RGB, possui três valores: Red, Green e Blue. A união destes valores dá forma a um pixel de uma cor específica e a união destes pixels dá forma a uma imagem. Porém, em se tratando de vídeos, como o exemplo citado, uma frequência de 144 fps reproduz 144 imagens no intervalo de 1 segundo. O processamento computacional envolvido em uma matriz de dados assim é gigante. Na computação as matrizes estão presentes em diversas aplicações, e a ideia de reprodução de imagem é apenas uma delas, citada para demonstrar o quão custosa pode ser uma operação utilizando-se de um conceito como este. As diferentes abordagens de matrizes como: soma, divisão e multiplicação, requerem diferentes abordagens de cálculos e organizações de dados para a obtenção de um melhor resultado. O custo de um cálculo desta grandeza é alto, e quanto maiores forem os dados de entrada, maior será o tempo necessário para realizá-lo.

No entando, o conceito de programação paralela concede uma melhora considerável quando o requisito é tempo de execução. O intuito é dividir o problema para que seja calculado parcialmente e paralelamente por diferentes processos, reduzindo o tempo gasto. Uma das abordagens da programação paralela se baseia na criação das chamadas *threads*. Cada *thread* executa uma parte do problema até que tenha um fim. Uma analogia seria a de divisão de correções de provas. Um número N de provas para 1 professor será mais rapidamente corrigido se for divido entre um número $M \neq 1$ de professores.

Este trabalho apresenta a implementação de um programa, construído na linguagem de programação C, que realiza a multiplição de duas matrizes quadradas. A abordagem utiliza a biblioteca Pthreads, disponibilizada também pelo C. O problema consiste em

gerar aleatoriamente duas matrizes quadradas, realizar o produto entre elas, salvar os dados em uma matriz destinada ao resultado e simular a solicitação de informações aos dados utilizando a *Main Thread*.

2 Desenvolvimento

2.1 Cálculo

Na matemática, o produto de duas matrizes $A \times B$ é definido se e somente se o número de colunas da matriz A for igual ao número de linhas da matriz B. Se A é uma matriz $m \times$n, e B uma matriz $n \times$p, então o resultado da equação resultará em uma matriz C de ordem $m \times$p. Cada elemento $c_{i,j}$ da matriz resultante é calculado a partir da soma dos produtos entre os elementos da i-ésima linha de A pela j-ésima coluna de B. A função pode ser generalizada por:

$$\sum_{r=1}^{n} a_{i,r} b_{r,j} = a_{i,1} b_{,j} + a_{i,2} b_{2,j} + \ldots + a_{i,n} b_{n,j}$$

Portanto, um algoritmo para realizar o produto de cada elemento de A por B pode ser descrito abaixo. A variável *resultado* armazena o resultado da operação, que pode ser adicionado à uma matriz resultante qualquer.

```
1     for (i = 0; i < linhas; i++) {
2         for (j = 0; j < colunas; j++) {
3             for (k = 0; k < linhas; k++){
4                 resultado[i][j] = resultado[i][j] + (a[i][k] * b[k][j]);
5             }
6         }
7     }
```

2.2 Código Serial

A abordagem serial para este problema não envolve muitas etapas. Em resumo, o programa precisa gerar e alocar as matrizes de forma dinâmica para então fazer o cáculo de cada elemento individualmente.

O programa *serial_matrix_calc.c* foi inicializado com as variáveis necessárias. O trecho de código abaixo indica quais.

```
1     int i, j, k, result = 0;
2     int size_matrix;
3     int **a;
4     int **b;
5     int **c;
```

```
6     double total_time;
7     clock_t start, end;
```

2.2.1 Variáveis

As variáveis i, j e k são contadores usados nos loops para realizar o cálculo. A variável *result* armazena o valor atual do elemento $c_{i,j}$ calculado. O tamanho do problema, ou ainda, o número de linhas/colunas (a matriz é quadrada) das matrizes geradas foi armazenado na variável *size_matrix*. Os ponteiros de ponteiros a, b e c representam respectivamente as matrizes A, B e C utilizadas como exemplo na sessão 2.1. A variável *total_time* representa o tempo total gasto pelo cálculo do programa, dado pela diferença entre as variáveis *end* e *start* do tipo *clock_t*.

2.2.2 Alocação dinâmica

O trecho de código abaixo mostra como foi feita a alocação dinâmica e o preenchimento das matrizes.

```
1     a = (int**)malloc(size_matrix * sizeof(int*));
2     b = (int**)malloc(size_matrix * sizeof(int*));
3     c = (int**)malloc(size_matrix * sizeof(int*));
4
5     for(i = 0; i < size_matrix; i++){
6         a[i] = (int*)malloc(size_matrix * sizeof(int));
7         b[i] = (int*)malloc(size_matrix * sizeof(int));
8         c[i] = (int*)malloc(size_matrix * sizeof(int));
9     }
10
11    srand(time(NULL));
12    for(i = 0; i < size_matrix; i++){
13        for (j = 0; j < size_matrix; j++) {
14            a[i][j] = rand() % 100;
15            b[i][j] = rand() % 100;
16            c[i][j] = -1;
17        }
18    }
```

A primeira alocação, feita entre as linhas 1 e 3, indica que cada posição do array de inteiros que tem tamanho *size_matrix* (ou seja, o número de linhas da matriz) armazena um outro array de inteiros. O laço *for* (linhas 5 à 9) pecorre cada elemento alocando também de forma dinâmica os novos arrays de inteiros.

A função *srand* é inicializada na linha 11, tendo como semente um valor nulo, e é utilizada para gerar valores randômicos. Os laços das linhas 12 e 13 percorrem *size_matrix*

vezes cada, ou seja, tem uma complexidade O(n^2). A cada iteração do laço da linha 13, dois valores aleatórios no intervalo de 0 à 100 (intervalo escolhido para facilitar os cálculos) é gerado e atribuído às matrizes *a* e *b*. A matriz *c* recebe em todos os seus elementos o valor -1, que posteriormente será substituído pelo resultado da multiplicação.

2.2.3 Cáculo da matriz e tempo de execução

Por definição em sala de aula, o tempo de execução do programa seria apenas o necessário para realizar os cálculos dos elementos da matriz. Portanto, as variáveis que representam o início e o fim do tempo foram postas uma linha antes e uma linha depois do laço de execução dos cálculos, respectivamente. O tempo total é calculado na linha 15, realizando a diferença entre o fim e o início, além de dividir o valor pelo clock do sistema.

```
1     start = clock();
2
3     for (i = 0; i < size_matrix; i++) {
4         for (j = 0; j < size_matrix; j++) {
5             for (k = 0; k < size_matrix; k++) {
6                 result += a[i][k] * b[k][j];
7             }
8             c[i][j] = result;
9             result = 0;
10        }
11    }
12
13    end = clock();
14
15    total_time = ((double) (end - start)) / CLOCKS_PER_SEC;
16
17    FILE *tempo;
18    tempo = fopen("tempo_de_exec_serial.txt", "a");
19    fprintf(tempo, "Problem size = %d ----- Runtime = %f\n", size_matrix,
          total_time);
20    fclose(tempo);
```

O trecho de código entre as linhas 3 e 11 tem complexidade O(n^3). Para cada elemento calculado, é necessária a soma entre *size_matrix* elementos, por isso o terceiro laço é adicionado. O cálculo da multiplicação é explicado na sessão 2.1.

Por fim, o tamanho do problema e seu tempo de execução são impressos em um arquivo de saída chamado *tempo_de_exec_serial.txt*.

2.2.4 Execução do código

Um *Shell Script* chamado *script_serial_matrix.sh* foi criado para compilar e executar o programa. São impressos alguns caracteres no arquivo gerado pelo programa. De antemão, o script remove e recria o arquivo para poder limpar o resultado de outras execuções anteriores.

```
1  rm tempo_de_exec_serial.txt
2  touch tempo_de_exec_serial.txt
3
4  gcc -g -Wall -o serial_matrix serial_matrix_calc.c
5
6  tentativas=4
7  for size in 50 100 200 400 #tamanho do problema
8  do
9          echo -e "=======================================\n" >> "
               tempo_de_exec_serial.txt"
10                 for tentativa in $(seq $tentativas)
11                 do
12                         echo -e `./serial_matrix $size`
13                 done
14                 echo -e "\n====================================" >> "
                       tempo_de_exec_serial.txt"
15 done
16 exit
```

São executadas 4 tentativas para cada tamanho de problema definido, que no caso são: 50, 100, 200 e 400. Os comandos *echo* são utilizados para fornecer uma melhor visualização da saída padrão do programa no arquivo *tempo_de_exec.txt*.

2.3 Código Paralelo

Para o programa paralelo, a abordagem escolhida foi a de fazer com que cada thread criada pegue um elemento $c_{i,j}$ aleatoriamente e o calcule. Assim, quando todos os elementos forem calculados, as threads param e o programa tem seu fim.

Foi definido que a thread *main* ficaria responsável por simular o acesso de dados enquanto os elementos da matriz C são calculados, tudo isso paralelamente. Enquanto as threads calculam os valores, a main os requisita e realiza uma cópia para uma outra matriz *D*. Caso o valor ainda não tenha sido calculado, a main não espera e então solicita um próximo.

As alocações dinâmicas e preenchimentos foram feitos exatamente iguais aos descritos na sessão 2.2.2, com a adição apenas da nova matriz *D*, tendo seus elementos

inicializados com -2.

Para fazer com que os pares de coordenadas na matriz C fossem sorteados, um array de tamanho *size_matriz* × *size_matrix* foi criado, sendo preenchido com todas as possibilidades de pares para o tamanho do problema solicitado, indo de (0,0) e ((size_matriz - 1), (size_matriz - 1)). Após a criação e preenchimento do array, um *shuffle* é realizado para que os pares fiquem dispostos de forma aleatória por todo o array. O trecho de código abaixo mostram como foi feito.

```
1  /* Dynamic allocation to stack*/
2  stack = (node *)malloc(sizeof(node));
3  /* Allocating auxiliar array to all possible (x,y)pairs */
4  v_coordinates = (int**)malloc(size_matrix * size_matrix * sizeof(int*));
5  for (i = 0; i < size_matrix * size_matrix; i++) {
6      v_coordinates[i] = (int*)malloc(2 * sizeof(int));
7  } /* endfor */
8
9  /* Dynamic allocation and fill to array of semaphores */
10 v_semaphores = (sem_t *)malloc(size_matrix * size_matrix * sizeof(sem_t));
11 for (i = 0; i < size_matrix * size_matrix; i++) {
12     sem_init(&v_semaphores[i], 0, 0);
13 }
14
15 /* Populating v_coordinates with size_matrix * size_matrix possible pairs
16 (in order) */
17 for (i = 0; i < size_matrix * size_matrix; i+=size_matrix) {
18     for (j = i; j < i + size_matrix; j++){
19         v_coordinates[j][0] = count_x;
20         v_coordinates[j][1] = count_y;
21         count_y++;
22     }
23     count_y = 0;
24     count_x++;
25 } /* endfor */
26
27 /* Shuffling the array of coordinates */
28 srand(time(NULL));
29 for (i = 0; i < size_matrix * size_matrix; i++) {
30     int first = rand() % (size_matrix * size_matrix);
31     int second = rand() % (size_matrix * size_matrix);
32
33     int x = v_coordinates[first][0];
34     int y = v_coordinates[first][1];
35
36     v_coordinates[first][0] = v_coordinates[second][0];
37     v_coordinates[first][1] = v_coordinates[second][1];
```

```
38
39      v_coordinates[second][0] = x;
40      v_coordinates[second][1] = y;
41  } /* endfor */
42
43  if(!stack){
44      printf("Error! No memory available.\n");
45      exit(1);
46  } else
47      startStack(stack);
48
49  /* Pushing all shuffled possibilities to the stack */
50  for (i = 0; i < size_matrix * size_matrix; i++) {
51      int x = v_coordinates[i][0];
52      int y = v_coordinates[i][1];
53      push(stack, x, y);
54  } /* endfor */
```

O array *v_semaphores* foi criado para armazenar um total de *size_matriz* × *size_matrix* semáforos, o que representa 1 semáforo para cada par possível na matriz *C*. Os semáforos são necessários para que a thread main verifique se o elemento que está representado em um índice *i* no array já foi calculado ou não. Essa necessidade se dá por conta da natureza da operação que os envolve. Quando um semáforo tem seu valor mudado, seja em UP ou DOWN, esta operação é feita de forma atômica, o que significa dizer que tem um início e fim, obrigatoriamente. A main fará requisições neste array, buscando índices aleatórios. Esta busca também ocorre de forma atômica. Sendo assim, não haverá inconsistência de dados entre a mudança do semáforo e a busca da thread main.

A segunda parte envolve uma estrutura de dados, e a escolhida foi a pilha. Cada elemento presente no array de coordenadas, já embaralhado, foi adicionado à pilha, gerando assim uma pilha embaralhada (linhas 50 à 54). Com esta abordagem, as threads selecionam elementos $c_{i,j}$ de forma aleatória. A pilha foi escolhida pela facilidade na remoção dos elementos, assim, cada pode remover um elemento e deixa que a próxima faça o mesmo. Desde modo, os elementos removidos não são calculados novamente, evitando redundância.

Após as alocações e preenchimentos das matrizes e da stack auxiliar, a abetura da região paralela é o próximo passo. O número de threads utilizadas para a execução do programa é resgatado via linha de comando, o que é feito automaticamente pelo script descrito na sessão 2.3.3.

```
1  thread_handles = malloc (thread_count * sizeof(pthread_t));
2  start = clock(); // start clock only for calculation
3
4  /* Creates thread 0 to thread_count-1 */
```

```
5  for (thread = 0; thread < thread_count; thread++)
6      pthread_create(&thread_handles[thread], NULL,
7          matrizCalc, (void*) thread);
8
9  /* main thread requests in matriz C */
10 srand(time(NULL));
11 int index, value, x, y;
12 do {
13     index = rand() % (size_matrix * size_matrix);
14     sem_getvalue(&v_semaphores[index], &value);
15
16     if (value == 1) {
17         y = index % size_matrix;
18         x = index/size_matrix;
19
20         if (c[x][y] == -1) {
21             printf("Wrong value (-1) on C matrix\n");
22             exit(0);
23         } else if(d[x][y] == -2) {
24             d[x][y] = c[x][y];
25         }
26     }
27 } while(threads_working > 0);
28
29 for (thread = 0; thread < thread_count; thread++)
30     pthread_join(thread_handles[thread], NULL);
31
32 end = clock(); // end clock
```

A segunda linha representa o início da contagem do tempo de execução, e a última, o seu fim, sendo sua diferença a representação do tempo total gasto pelo cálculo das threads. As threads são criadas no loop da linha 5, e são finalizadas em um outro loop, presente na linha 29. Entre a criação e fim das threas, da linha 9 à linha 27, existe um loop onde a main fará a requisição de dados na matriz C para realizar a cópia para a matriz D. A cópia consiste em 3 partes:

(i) Gerar um índice aleatório entre 0 e $size\_matriz \times size\_matrix$;

(ii) Selecionar no array de semáforos o índice sorteado e verificar se seu valor é 1, caso seja, as coordenadas x e y são calculadas a partir deste índice;

(iii) Se o valor da matriz C nas posições x e y for igual a -1, significa que houve um problema no cálculo do elemento (o programa diz que já alguma thread já calculou, mas na verdade isto não ocorreu, pois a matriz C ainda possui seu valor inicial, e, em casos assim, o speedup do programa é 0);

(iv) Caso o valor da matriz C nas posições x e y seja diferente de -1, uma nova verificação é feita acerca de assegurar que o elemento já não foi copiado anteriormente;

(v) Caso tudo ocorra normalmente, o valor da matriz D nas posições x e y recebe o valor da matriz C nas mesmas coordenadas

O laço *while* só é finalizado quando a variável *threads_working* atinge o valor de 0. Ela é inicializada com o valor de threads que serão criadas no programa, assim, a cada thread que termina seu cálculo, o valor é decrementado em 1, chegando a 0 quando todas finalizarem.

2.3.1 Função das threads

O trecho de código apresenta a função *matrizCalc()* que será executada por cada thread.

```
1  void *matrizCalc(void* rank) {
2      int k, local_x, local_y, mult_result = 0;
3      node *local_tmp;
4
5      srand(time(NULL));
6
7      // Interruped only when the stack is empty
8      while(true){
9          pthread_mutex_lock(&mutex); // lock to pick up the stack element
10         if (stack->prox == NULL){
11             pthread_mutex_unlock(&mutex);
12             break;
13         } // empty stack?
14
15         local_tmp = pop(stack);
16         pthread_mutex_unlock(&mutex);
17
18         local_x = local_tmp->x;
19         local_y = local_tmp->y;
20
21         // individual calculation to each pair
22         for (k = 0; k < size_matrix; k++){
23             mult_result += a[local_x][k] * b[k][local_y];
24         }
25
26         c[local_x][local_y] = mult_result; // attribution to C matrix
27         mult_result = 0; // reset to the next iteration
28
29         // post (+1) at the corresponding semaphore
```

```
30            sem_post(&v_semaphores[local_x*size_matrix+local_y]);
31        }
32
33        pthread_mutex_lock(&mutex_threads_working);
34        threads_working--;
35        pthread_mutex_unlock(&mutex_threads_working);
36
37        return NULL;
38    }
```

O laço principal da função calcula um elemento $c_{i,j}$ por vez, e é finalizado quando a pilha de remoções dos pares de coordenadas é vazia, o que significa que todos os elementos já foram calculados (verificação na linha 10). Caso ainda exista algum elemento a ser calculado, a thread bloqueia a pilha[1], recupera o primeiro elemento da pilha e libera a região. O valor é armazenado e as coordenadas de x e y são salvas. O laço na linha 22 realiza o cálculo do elemento e o atribui à variável *mult_result* que é zerada a cada iteração do *while*. O valor de *mult_result* é atribuído à sua posição (x,y) em C e o seu índice no array de semáforos é alterado para 1, informando a main que o elemento foi calculado com sucesso. O índice é definido por:

$$local\_x \times size\_matrix + local\_y$$

Este índice representa a posição linear do par de coordenadas x e y em um array unidimensional, em ordem crescente de valores. Ou seja, em um array de semáforos que tem como tamanho de problema uma matriz 10×10, o elemento do índice 0 representa o par (0,0), o índice 10 representa o par (0,10), o índice 11 o par (1,0), e assim sucessivamente.

Quando não existem mais valores a serem calculados, a thread termina seu laço e decrementa a variável *threads_working* para informar ao main que já acabou.

2.3.2 Regiões críticas

Para que as threads possam resgatar os valores da pilha, é necessária a criação de um bloqueio. Como a operação envolve mudança na estrutura da pilha, já que o elemento é removido para não ser mais resgatado, um *mutex* foi criado especialmente para esta operação de remoção o primeiro elemento da pilha. A thread chama a função *pthread_mutex_lock()* antes de solicitar à stack que realize o *pop()* do seu primeiro elemento. O elemento é salvo localmente e então a função *pthread_mutex_unlock()* é chamada para que as threads seguintes realizem a mesma operação. O código abaixo mostra a região crítica na função das threads.

[1] Explicado na sessão 2.3.2

```
1     while(true){
2         pthread_mutex_lock(&mutex); // lock to pick up the stack element
3         if (stack->prox == NULL){
4             pthread_mutex_unlock(&mutex);
5             break;
6         } // empty stack?
7
8         local_tmp = pop(stack);
9         pthread_mutex_unlock(&mutex);
```

Além da região crítica na execução do *pop()* da pilha, uma nova região está presente no decremento da variável *threads_working*. Portanto, uma nova variável *mutex_threads_working* do tipo *pthread_mutex_t* foi criada. A cada thread que finaliza seu trabalho, é feito o lock na região, a variável global é decrementada, e o unlock é realizado, garantindo que apenas uma thread por vez faça o decremento, até que o valor chegue a 0. O trecho de código abaixo mostra a região citada.

```
1     pthread_mutex_lock(&mutex_threads_working);
2     threads_working--;
3     pthread_mutex_unlock(&mutex_threads_working);
```

2.3.3 Execução do código

De forma análoga ao script que executa o programa serial na sessão 2.2.4, um script *script_matrix.sh* foi criado para gerar os resultados. A diferença se dá pelo laço representando as threads que são utilizadas (8, 16, 32 e 63). Além disso, o programa recebe dois parâmetros por linha de comando: número de threads e tamanho do problema, nesta ordem.

```
1  rm tempo_de_exec.txt
2  touch tempo_de_exec.txt
3
4  gcc -g -Wall -o matrix matrix_calc.c -lpthread
5
6  for size in 50 100 200 400 #tamanho do problema
7  do
8          echo -e "=======================================\n" >> "
               tempo_de_exec.txt"
9                  for thread in 8 16 32 63 #numeros de threads utilizadas
10                 do
11                         for tentativa in $(seq $tentativas)
12                         do
13                                 echo -e `./matrix $thread $size`
14                         done
```

```
15                          echo -e " " >> "tempo_de_exec.txt"
16                  done
17                  echo -e "\n==================================" >> "
                        tempo_de_exec.txt"
18  done
19
20  exit
```

3 Resultados e Análise

3.1 Runtime

Como descrito no capítulo 2, cada programa, serial e paralelo, foi executado um total de 4 vezes para cada tamanho de problema definido. O programa paralelo obteve mais execuções por conta do número de threads diferentes em cada tamanho de problema, gerando um total de 64 execuções, enquanto o serial gerou 16.

3.1.1 Média de tempo de execução

Abaixo é descrita uma tabela que exibe a média de execução para cada problema nos algoritmos serial e paralelo.

Tamanho do Problema	Number of Threads				
	Serial	8	16	32	63
50	0,00050175	0,045889	0,0457985	0,04565	0,050504
100	0,00502975	0,6254845	0,62674325	0,62863075	0,6394575
200	0,03690125	9,836772	9,7703425	9,7801545	8,5476485
400	0,31412475	155,6944305	149,35500075	151,27224775	168,46290375

3.2 SpeedUp e Eficiência

Como visto na tabela da sessão 3.1.1, o tempos do algoritmo serial foram, em todos os casos, melhores que os da implementação paralela. Isto implica em dizer que o algoritmo não possui SpeedUp ou Eficiência.

4 Conclusão

A ideia inicial da implementação do algoritmo sugeria uma boa abordagem, levando em consideração a busca aleatória por elementos na matriz C. Entretanto, o algoritmo se mostrou ineficaz. O trabalho exclusivo que a main faz na requisição de elementos na matriz C é uma das causas disto. Caso removido do código, este trecho implica em uma redução de quase 50% no tempo gasto.

No entando, existem outras causas na obtenção de um resultado desta natureza. Provavelmente, a busca na pilha e as operações realizadas, como o *pop()*, também elevem o tempo de execução, por existir manipulação de ponteiros na função.

Em resumo, o algoritmo escolhido não é escalável, sendo ineficiente, ao menos, para os tamanhos de problema escolhidos. Talvez uma implementação que não aborde uma busca randômica por elementos consiga obter um SpeedUp considerável