Fabrício Mady **1215080268**

Multiplicação de Matrizes – OpenMP

1 – Código em C++:

MatrizesOpenMP (Escopo Global)

```
1   #include <iostream>
2   #include <omp.h>
3   #include <cstdlib>
4   #include <cmath>
5   #include <ctime>
6
7   #define TAM  5000
8
9   using namespace std;
10
11
12  int main() {
13
14      int i, j,k,*a,*b,*c,tmp;
15      int n, m, p;
16
17      n = m = p = TAM;
18
19      clock_t tempo_inicial, tempo_final;
20
21      a = (int*)malloc(n * p * sizeof(int));
22      b = (int*)malloc(p * m * sizeof(int));
23      c = (int*)malloc(n * m * sizeof(int));
24
25      for (i = 0; i < TAM; i++) {
26          for (j = 0; j < TAM; j++) {
27              *(a + (i * n + j)) = rand() % 10;
28              *(b + (i * n + j)) = rand() % 10;
29              *(c + (i * n + j)) = 0;
30          }
31      }
32
33      tempo_inicial = clock();
34
35      omp_set_num_threads(4);
36
```

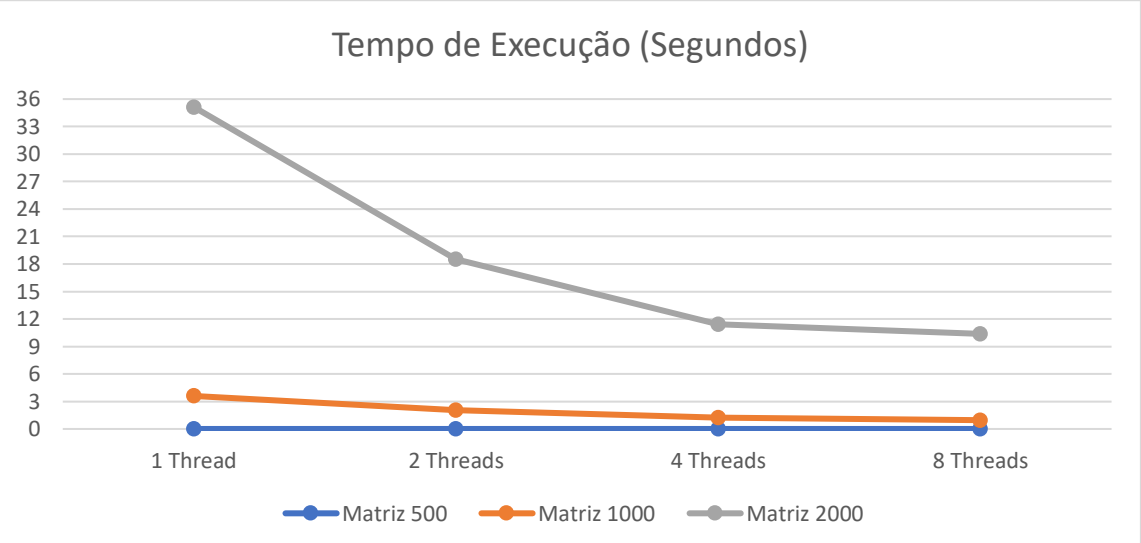
```
36
37  #pragma omp parallel for private(tmp, i, j, k) shared (n, m, p)
38      for (i = 0; i < n; i++) {
39          for (j = 0; j < m; j++) {
40              tmp = 0;
41              for (k = 0; k < p; k++) {
42                  tmp += (*(a + (i * n + k))) * (*(b + (k * p + j)));
43              }
44              *(c + (i * n + j)) = tmp;
45          }
46      }
47
48      tempo_final = clock();
49
50      cout << fixed;
51      cout.precision(6);
52      cout << "execucao: " << (tempo_final - tempo_inicial) / ((double)CLOCKS_PER_SEC);
53
54  /*
55      for (i = 0; i < TAM; i++) {
56          for (j = 0; j < TAM; j++) {
57              cout << *(c + (i * TAM + j));
58          }
59      }  */
60
61      return 0;
62
63  }
```

2 – Avaliação dos Resultados:

THREADS	TAMANHO MATRIZ	TEMPO DE EXECUÇÃO MULTIPLICAÇÃO (5X EXECUTADO)					MÉDIA
1	500	0.437000,	0.480000,	0.499000,	0.486000,	0.467000	0.4738
2	500	0.232000,	0.213000,	0.212000,	0.260000,	0.234000	0.2302
4	500	0.158000,	0.110000,	0.128000,	0.117000,	0.143000	0.1312
8	500	0.110000,	0.099000,	0.099000,	0.119000,	0.120000	0.1094

THREADS	TAMANHO MATRIZ	TEMPO DE EXECUÇÃO MULTIPLICAÇÃO (5X EXECUTADO)					MÉDIA
1	1000	3.824000,	3.704000,	3.591000,	3.560000,	3.614000	3.6586
2	1000	2.143000,	2.225000,	1.983000,	2.106000,	1.968000	2.0850
4	1000	1.376000,	1.154000,	1.362000,	1.035000,	1.296000	1.2446
8	1000	1.006000,	0.950000,	0.970000,	0.979000,	0.930000	0.9670

THREADS	TAMANHO MATRIZ	TEMPO DE EXECUÇÃO MULTIPLICAÇÃO (5X EXECUTADO)					MÉDIA
1	2000	31.935000,	31.266000,	31.748000,	31.480000,	31.251000	31.5360
2	2000	15.919000,	16.083000,	15.945000,	17.149000,	17.109000	16.4410
4	2000	10.866000,	10.450000,	10.381000,	9.640000,	9.657000	10.1988
8	2000	9.529000,	9.188000,	9.299000,	9.757000,	9.359000	9.4264



O que se pode perceber é que, paralelizar o processo da multiplicação da matriz, traz melhores resultados a partir de matrizes grandes. Como é perceptível no gráfico acima, o tempo ganho paralelizando matrizes de valores menores, é basicamente imperceptível quando se compara a 1 thread (sequencial). O programa foi executado em um processador quad-core, ou seja, para sua melhor otimização o ideal é apenas utilizar 4 threads, visto que, acima disso, não se percebe melhora em desempenho (tempo de execução).

3 – Descrição dos processos do OpenMP para resolução do problema:

```
omp_set_num_threads(8);
```

De certa forma, o processo de utilização da OpenMP é simples. Começando pela figura acima, esse comando foi necessário para poder se definir manualmente a quantidade de threads para cada execução, sendo que, se não utilizar o mesmo, por definição, o compilador utiliza todos os threads do processador.

```
#pragma omp parallel for private(tmp, i, j, k) shared (n, m, p)
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {
            tmp = 0;
            for (k = 0; k < p; k++) {
                tmp += (*(a + (i * n + k))) * (*(b + (k * p + j)));
            }
            *(c + (i * n + j)) = tmp;
        }
    }
```

Nesta segunda imagem, é o processo de multiplicação de matrizes propriamente dito. A primeira linha, é declarado o comando #pragma omp parallel, este comando cria a região paralela, onde todos os threads acessam o mesmo endereço de memória das variáveis dessa região.

Em seguida, é declarado for private (tmp,i,j,k), essa declaração faz com que seja gerado para cada thread uma cópia de cada variável, onde apenas o mesmo o acessa. Essa declaração também é feita para não se ter o “overhead” da condição de corrida (como já visto em S.O).

Logo após, é declarado shared (n,m,p), que se quer dizer que, as variáveis (n,m,p) podem ser acessadas por todos os threads normalmente, neste sentido não se tem impacto visto que os valores são iguais a constante de tamanho da matriz definida inicialmente no código.

Por fim, é feito normalmente o cálculo, onde cada thread fica responsável por uma linha da multiplicação da matriz em si.