

Aluno: *Sérgio Luciano de Oliveira Soares*

RA: 263560

Relatório para disciplina FT077A - Processamento de Alto Desempenho.

Prof. André Leon S. Gradvohl, Dr.

07/05/2021

SUMÁRIO

1. INTRODUÇÃO.....	3
2. METODOLOGIA.....	4
3. DESENVOLVIMENTO.....	9
4. RESULTADOS E GRÁFICOS	13
5. RESPOSTAS SOLICITADAS.....	17
6. OBSERVAÇÕES.....	18

1. INTRODUÇÃO

Com base no problema proposto anteriormente realizado no laboratório 01 em PThreads e agora utilizando a mesma otimização de códigos em OpenMp, o mesmo consiste em criar um programa serial e um programa paralelo com OpenMP que calcule a operação matricial $D = A * B + C$, onde todas as matrizes (A, B, C e D) têm dimensões $n \times n$.

2. METODOLOGIA

Para esse exercício foi realizado apenas as adequações do programa desenvolvido anteriormente em PThreads reutilizando-se todas as otimizações e procedimentos que já haviam sido desenvolvidas no trabalho anterior, entretanto utilizando os recursos solicitados da tarefa 02 em OpenMP.

3. DESENVOLVIMENTO

Otimizações utilizadas para o cálculo:

$$D = A*B+C = C+A*B$$

soma de matrizes: $s_{ij} = a_{ij} + b_{ij}$

multiplicação de matrizes: $m_{ik} = a_{i1} \cdot b_{1k} + a_{i2} \cdot b_{2k} + \dots + a_{in} \cdot b_{nk}$

Para o cálculo do resultado D ($=A*B+C$), temos a fórmula para cada elemento:

$$D_{ik} = C_{i1} + A_{i1} \cdot B_{1k} + A_{i2} \cdot B_{2k} + \dots + A_{in} \cdot B_{nk}$$

A matriz foi alocada em uma única etapa (como um vetor).

O acesso aos elementos para o loop acima (1..n) causa o acesso aos elementos em B de forma não sequencial na memória.

Como o acesso a posições próximas de memória são mais interessantes, foi utilizada a abordagem de calcular a matriz transposta de B, de forma que o loop cause o acesso aos elementos de B de forma sequencial, assim como em A.

B_t = transposta de B

Com isso, temos:

$$D_{ik} = C_{i1} + A_{i1} \cdot B_{k1} + A_{i2} \cdot B_{k2} + \dots + A_{in} \cdot B_{kn}$$

Essa abordagem permitiu que o cálculo fosse feito sem que fossem criadas condições de corrida, com apenas a necessidade de uma barreira.

Se tivesse sido calculado $A*B$ para a matriz inteira e depois calculado $(A*B)+C$, haveria necessidade da criação das threads duas vezes (uma para a multiplicação, e outra para a adição), com o uso de uma barreira para cada etapa.

O uso da diretiva *collapse* foi dispensado uma vez que o programa desenvolvido anteriormente já implementava um laço único para tratar a matriz por elementos de forma sequencial, independente da linha e coluna do elemento, ao invés de tratar a matriz com um laço para as linhas e outro para as colunas.

O código apenas foi alterado para eliminar o controle de carga para cada thread que era realizado, passando a implementar apenas um laço com todos os elementos da matriz.

A função de cálculo foi duplicada no código apenas para garantir que no cálculo sequencial (1 thread) não houvesse sobrecarga gerada pelas diretivas do *OpenMP* com a definição de uma única thread (*num_threads(1)*).

O código todo foi feito em um único arquivo (*exerc2_OpenMP.c*).

Para compilação:

```
gcc -fopenmp exerc2_OpenMP.c -o exerc2_OpenMP
```

https://github.com/s263560/pad/blob/master/OpenMP/exerc2_OpenMP.c

A validação do algoritmo foi feita com $n=10$, com 3 threads, pegando as matrizes produzidas pelo programa e refazendo o cálculo no excel, com o resultado do programa ficando exatamente igual ao cálculo executado no excel.

4. RESULTADOS E GRÁFICOS

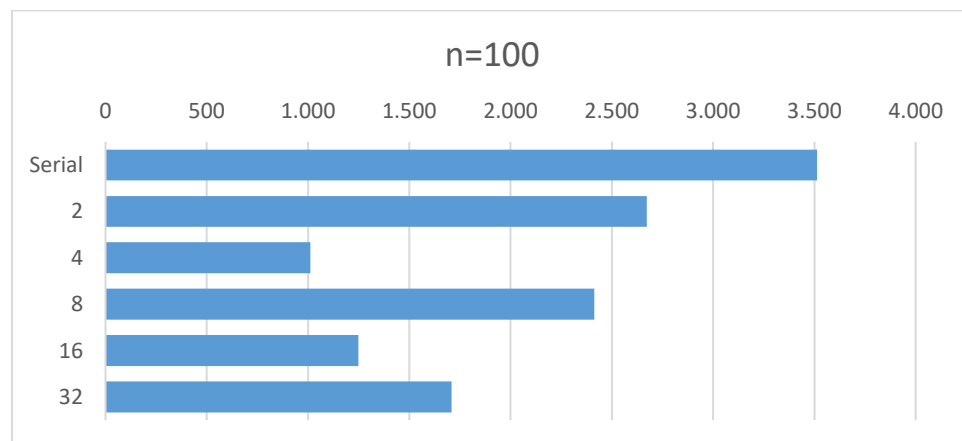
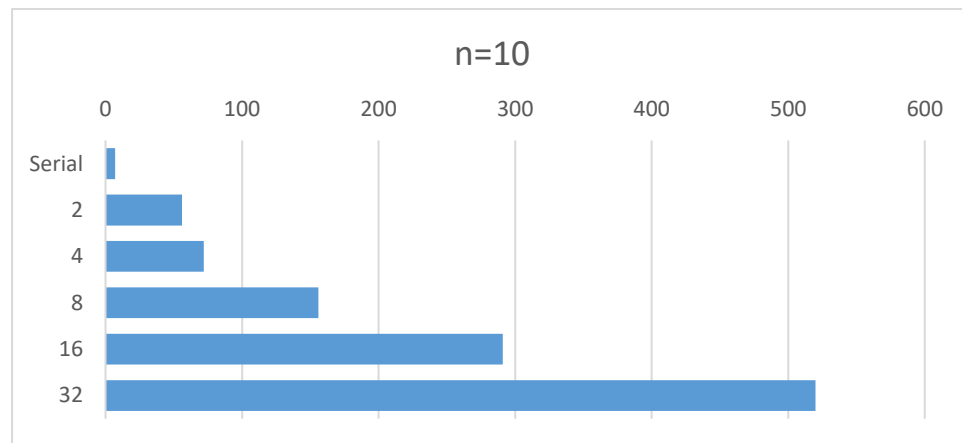
A implementação do paralelismo com o *OpenMP* no código foi bem mais simples que a implementação inicial feita com *pthread*s, uma vez que foi necessário apenas o uso de duas diretivas para paralelizar o laço principal do programa, sem a necessidade de realizar o cálculo de distribuição de carga para cada thread, como havia sido feito no programa anterior.

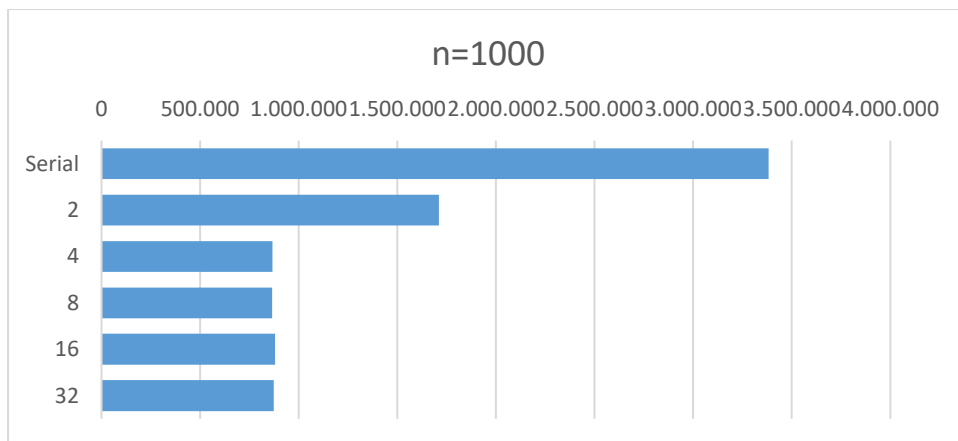
Calcule o tempo de execução do programa serial para matrizes de tamanho $n \times n$, onde $n = 10, 100$ e 1000 .

Calcule o tempo de execução do programa paralelo para matrizes de tamanho $n \times n$, onde $n = 10, 100$ e 1000 , cada uma com 2, 4, 8, 16 e 32 threads.

Tempos de execução em microssegundos:

n	Threads					
	Serial	2	4	8	16	32
10	7	56	72	156	291	520
100	3.513	2.672	1.011	2.413	1.249	1.709
1000	3.381.772	1.711.305	866.334	864.944	879.532	872.504





5. Respostas solicitadas

Há necessidade de sincronização entre as threads para resolver as operações?

Não. Do modo como o algoritmo foi implementado, não existem condições de corrida, não necessitando então do uso da diretiva *reduction*. Apenas foi necessária a barreira para aguardar a execução de todas as threads, implementada automaticamente pela diretiva *parallel* do OpenMP.

Qual foi o speedup em relação ao programa serial em cada uma das execuções?

Speedup:

n	Threads				
	2	4	8	16	32
10	0,13	0,10	0,04	0,02	0,013
100	1,31	3,47	1,46	2,81	2,06
1000	1,98	3,90	3,91	3,84	3,88

Houve algum caso em que não houve speedup em relação ao programa serial? Se houve, qual a razão para isso?

Sim.

Para $n=10$, não houve speedup ($\text{speedup} < 1$) em nenhuma situação.

A razão é que a quantidade de processamento para o cálculo do resultado é menor que o overhead de criação e controle das tarefas.

Para $n=100$, esse overhead tornou-se maior a partir do uso de 8 threads, onde o tempo de overhead passou a estar equilibrado com o tempo ganho pelo uso de mais threads, causando resultados de speedup que não estão diretamente proporcionais à quantidade de threads utilizadas – nem para mais, nem para menos.

Para $n=1000$, o speedup se manteve constante a partir de 4 threads, provavelmente por ter atingido o limite de threads reais que o host podia executar simultaneamente.

6. OBSERVAÇÕES

Para resolver tal exercício com segurança precisei das aulas 1 e 2, porém apenas as diretivas apresentadas na aula 1 foram utilizadas para a implementação.