

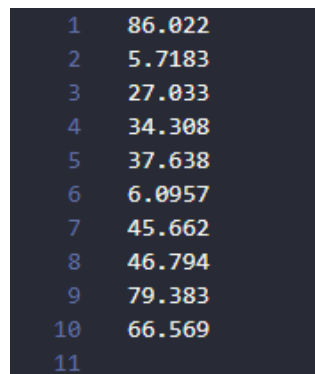
## 1 Introdução

Esse trabalho é a resolução proposta para o Exercício de Programação 1, da matéria de Organização e Arquitetura de Computadores II (ACH2055), ministrada pelo Prof. Dr. Clodoaldo A. de Moraes Lima.

Todos os códigos desenvolvidos podem ser vistos no repositório do GitHub, na url <https://github.com/victorIsDeving/sistemas-de-informacao/tree/main/oac2-ACH2055/ep2>.

## 2 Objetivo

O trabalho visa utilizar a linguagem C para criar um programa que vai comparar o tempo de execução para funções de ordenação realizadas com paralelização e sem paralelização. O programa recebe como *input* um arquivo de texto *.txt* com uma lista de números e gere um *output* dos números desta lista ordenados. Um exemplo de *input* é mostrado na Figura 1.



```
1 86.022
2 5.7183
3 27.033
4 34.308
5 37.638
6 6.0957
7 45.662
8 46.794
9 79.383
10 66.569
11
```

Figura 1: Exemplo de um arquivo de texto que pode ser usado como *input*.

O *output* também deve ser escrito no mesmo arquivo de texto utilizado como *input*, depois da lista original.

Dois métodos de ordenação devem ser implementados, um com complexidade  $O(n^2)$  e outro com  $O(n \log n)$ .

## 3 Metodologia

O exercício foi feito com o Visual Studio Code [4], disponibilizado pela Microsoft.

Os métodos de ordenamento feitos para esse trabalho, de acordo com o proposto pelo professor, são o *Bubble Sort* e o *Quick Sort*.

A ferramenta utilizada para a paralelização multithread foi o OpenMP (Open Multi-Processing) [1], um suporte multiplataforma, para C, C++ e Fortran, que disponibiliza diversas opções de paralelização multithread.

## 4 Desenvolvimento

O exercício foi feito primeiro para ser resolvido em thread única (programa sequencial), exposto no arquivo SEM-PARALELIZACAO.C, e depois feito com paralelização multithread (4 threads paralelas), exposto no arquivo COM-PARALELIZACAO.C, a fim de comparar os tempos de execução de cada resolução.

## 4.1 Paralelização do Bubble Sort

A paralelização do Bubble Sort foi feita em duas fases, a fase com iteração de número par e a fase de iteração com número ímpar[5]. A ideia é a de fazer o sistema fazer comparações paralelas e trocas paralelamente em cada iteração. O código desenvolvido foi inspirado no pseudo código ilustrado na Figura 2.

**PARALLEL BUBBLE SORT (A)**

1. For  $k = 0$  to  $n-2$
2. If  $k$  is even then
3.   for  $i = 0$  to  $(n/2)-1$  do in parallel
4.     If  $A[2i] > A[2i+1]$  then
5.       Exchange  $A[2i] \leftrightarrow A[2i+1]$
6. Else
7.   for  $i = 0$  to  $(n/2)-2$  do in parallel
8.     If  $A[2i+1] > A[2i+2]$  then
9.       Exchange  $A[2i+1] \leftrightarrow A[2i+2]$
10. Next  $k$

Figura 2: Pseudocódigo em que o código do Bubble Sort paralelizado foi inspirado.[5]

A ideia foi de usar o comando `#pragma omp parallel for` para paralelizar as interações da Array que iriam comparar e substituir os elementos conforme fosse necessário. Criando, assim, duas "zonas" de paralelização, uma para as iterações de número ímpar e outra para as iterações de número par. As zonas de paralelização não se sobrepõem. Cada uma inicia e termina dentro de cada iteração "mãe, conforme seja condicionado pela condicional de iteração ímpar/par.

## 4.2 Paralelização do Quick Sort

O código do QuickSort foi feito com base em um código do GitHub [3]. A estratégia consiste em criar uma zona de paralelização para cada chamada recursiva da função do Quick Sort, sendo que cada chamada terá sua própria cópia privada do endereço da array, do limite inferior e do limite superior da sub array. Por isso o uso do comando `#pragma omp task firstprivate(a,left,pi)` da biblioteca do OpenMP.

## 5 Conclusões

As Tabelas 1 e 2 mostram os tempos de execução, em segundos, para diferentes tamanhos de Array. Respectivamente, se referem aos tempos sem paralelização e com paralelização. O tempo de execução do Bubble Sort não foi obtido para a array de tamanho 10.000.000 (dez milhões), o computador em que o trabalho foi desenvolvido não conseguiu gerar o resultado para esse caso de teste.

SEM PARALELIZAÇÃO		
Tamanho	Bubble Sort	Quick Sort
10	0,000	0,000
30	0,000	0,000
50	0,000	0,000
100	0,010	0,000
1000	0,050	0,000
100000	50,749	0,029
1000000	5136,5660	0,289
10000000	n.a.	5,748

Tabela 1: Tabela com os tempos de execução obtidos, em segundos, nos testes de velocidade dos programas SEM paralelização.

COM PARALELIZAÇÃO		
Tamanho	Bubble Sort	Quick Sort
10	0,002	0,002
30	0,004	0,001
50	0,005	0,002
100	0,008	0,002
1000	0,078	0,002
100000	23,204	0,200
1000000	1752,7150	6,651
10000000	n.a.	557,1

Tabela 2: Tabela com os tempos de execução obtidos, em segundos, nos testes de velocidade dos programas COM paralelização.

Nos dados é possível visualizar que houve um ganho significativo com a paralelização do Bubble Sort. Para a array de tamanho 100.000 (cem mil), houve um ganho de quase 54% do tempo, caindo de 50,749 segundos para 23,204 segundos. Para a array de tamanho 1.000.000 (um milhão), o ganho foi ainda mais significativo, com uma redução de quase 66% do tempo de execução, de 5.136,5660 segundos para 1.752,7150 segundos. A diferença entre os tempos de execução com e sem paralelização do Bubble Sort pode ser visualizada no gráfico da Figura 3. Com o gráfico podemos visualizar que, para arrays pequenas, o Bubble Sort sem paralelização é mais eficiente, mas conforme a array cresce o Bubble Sort paralelizado passa a ser mais eficiente que o não paralelizado, com uma diferença cada vez maior de um para o outro quanto maior for a array. O ponto em que o algoritmo paralelizado passa a ser mais eficiente é, aproximadamente, para arrays de tamanho 100.000 (cem mil).



Figura 3: Gráfico comparando velocidades para ordenação com Bubble Sort de listas de diferentes tamanhos com e sem paralelização.

No Quick Sort houve um possível problema de implementação da paralelização que não foi possível de ser solucionado. Ele mostra dados inversos ao Bubble Sort, ou seja, ficou mais lento conforme aumenta o tamanho da array. Para a array de tamanho 1.000.000 (um milhão), houve uma perda de quase 2300% do tempo, aumentando de 0,289 segundos para 6,651 segundos. Para a array de tamanho 10.000.000 (dez milhões), a perda foi de quase 9700% do tempo de execução, de 5,748 segundos para 557,1 segundos. Um resultado muito diferente do esperado para a implementação da paralelização do algoritmo do Quick Sort, que seria o de um ganho de tempo que alcançaria um patamar conforme aumentava o tamanho da array. A

diferença entre os tempos de execução com e sem paralelização do Quick pode ser visualizada no gráfico da Figura 4.



Figura 4: Gráfico comparando velocidades para ordenação com Quick Sort de listas de diferentes tamanhos com e sem paralelização.

Foi feita uma pesquisa por uma implantação melhor do Quick Sort, mas que não foi possível de ser implementada com sucesso a tempo de entrega do trabalho. Uma possível solução seria mudar a implementação do Quick Sort para uma com múltiplos pivôs [2], usando as threads como uma forma de divisão da array principal, que seriam ordenadas separadamente, e depois juntadas de forma a manter a ordenação.

Uma outra possibilidade seria a atribuição de trabalhos de processos em árvore [6], onde a array seria quebrada de acordo com o número de threads e cada thread realizaria um quicksort próprio.

## Referências

- [1] OpenMP ARB. *OpenMP API specification for parallel programming*. URL: <https://www.openmp.org/>.
- [2] J. Bragamonte, F. Morales Antunes e B. Silveira Neves. «PARALELIZAÇÃO DO ALGORITMO QUICK-SORT MULTI-PIVOT COM A UTILIZAÇÃO DE THREADS». Em: *SIEPE 10.2* (2020).
- [3] koszio. *QuickSort-with-OpenMP*. URL: <https://github.com/koszio/QuickSort-with-OpenMP/blob/master/quickSortOpenMP.c>.
- [4] Microsoft. *Visual Studio Code*. URL: <https://code.visualstudio.com/>.
- [5] Rashid Bin Muhammad. *Bubble Sort*. URL: <https://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/bubbleSort.htm>.
- [6] Fernando Silva. *Algoritmos Paralelos - ordenação*. URL: <https://www.dcc.fc.up.pt/~ricroc/aulas/1011/ppd/apontamentos/sorting.pdf>.