Trabalho Final - Computação Paralela
BucketSort (Sequencial e Paralelizado)

Daniel Rubio Camargo | RA: 10408823
João Pedro Mascaro Baccelli | RA: 10224004

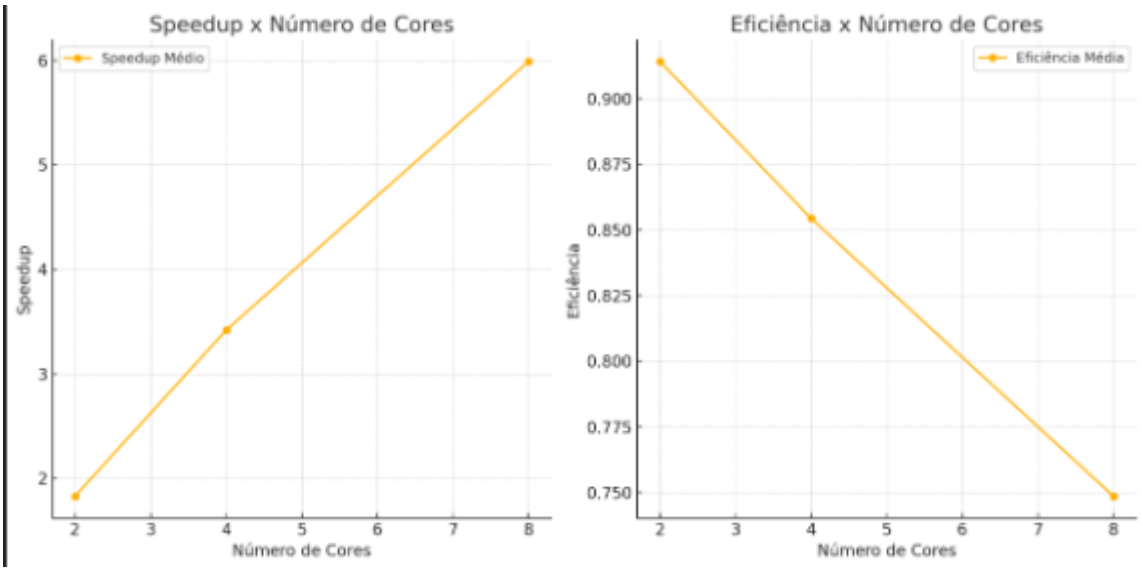
- **Tabela de tempo (em segundos)**

Quantidade de Strings	Tempo sequencial	Tempo paralelo (2 cores)	Tempo paralelo (4 cores)	Tempo paralelo (8 cores)
1.000	0.6	0.8	0.83	0.99
100.000	128.25	71.25	36.64	22.73
500.000	402.12	157.27	76.02	46.32
1.000.000	695.10	315.05	167.17	77.01

- **Tabela de eficiência**

Quantidade de Strings	Eficiência (2 cores)	Eficiência (4 cores)	Eficiência (8 cores)
1000	0.374	0.180	0.075
100.000	0.9	0.875	0.705
500.000	1.278	1.322	1.085
1.000.000	1.103	1.039	1.128

- **Gráfico de Speedup e Gráfico de Eficiência**



- **Texto comparativo**

Os dados demonstram que o sistema apresenta boa escalabilidade e eficiência em problemas maiores e até 4 núcleos, mas sofre perdas significativas em cenários com maior paralelismo e menor carga de trabalho. Isso evidencia a importância de balancear o uso de recursos paralelos com o tamanho do problema para garantir um desempenho eficiente. Ajustes na implementação do código poderiam melhorar a eficiência em casos de alta paralelização.

- **Discussão sobre a solução paralela**

Para resolvermos o problema do BucketSort paralelo, tivemos que analisar em qual parte do código seria inserida a paralelização (onde não haveria dependências ou outros tipos de problemas). Achamos viável inserir a paralelização em 2 partes do código, todas na própria função bucket_sort, onde praticamente tudo acontece. Projetamos o algoritmo para otimizar duas etapas principais: a distribuição das strings nos baldes e a ordenação dos baldes globais. A distribuição foi paralelizada utilizando baldes locais para evitar condição de corrida, e os dados foram mesclados nos baldes globais em uma seção crítica para então serem manipulados separadamente por cada thread.

- **Parte do código fonte paralelo**

```
#pragma omp parallel for // cada core faz a ordenação
for (i = 0; i < nbuckets; i++) {
    sort(a, &buckets[i]);
}

return returns;
```

Dentro da função “bucket_sort” teremos a parte em que serão ordenados os elementos de cada “bucket”. Como cada “bucket” consegue realizar as ordenações de forma independente, ou seja, o índice “i” usado no “for” será independente do seu próximo e de seu antecessor, é possível paralelizar, fazendo com que cada core ordene um “bucket” específico. Para isso chamamos o “#pragma omp parallel for” já que teremos as ordenações das iterações como grãos.

- **Bibliografia usada**

- Aulas no Moodle;
- BIANCHINI, Calebe, et al. Livro Programação Paralela e Distribuída, Casa do Código, Alura;
- Paralelismo de tarefas utilizando OpenMP 4.5 - Capítulo 6 - ResearchGate - por Calebe Bianchini, et al;
- Algoritmos de Ordenação - MAC5705, IME-USP.