MergeSort

Speedup, Eficiência e Escalabilidade da Versão Paralela

Sumário

Essa apresentação está dividida nos seguintes tópicos:

1. Ordenação de números utilizando MergeSort
2. Algoritmo Serial do MergeSort
3. Algoritmo Paralelo do MergeSort
4. Corretude do Algoritmo Serial do MergeSort
5. Corretude do Algoritmo Paralelo do MergeSort
6. Análise de Speedup, Eficiência e Escalabilidade
7. Considerações Finais

Ordenação de números utilizando MergeSort

Ordenação de números:

- Problema bem conhecido e abordado na área de computação
- Reposicionar os elementos em um vetor de forma que alguma ordem seja obtida, comumente a ordem não-decrescente
- Diversas aplicações (por exemplo: Busca Binária)

Ideia Geral do MergeSort

- Lema *“Dividir e Conquistar”*
- Natureza Recursiva
- Alto gasto de memória
- $O(n \cdot log(n))$ (Pior, médio e melhor casos)
- Diversas versões, cada uma com suas particularidades (Tanto Serial quanto Paralelo)

Algoritmo Serial do MergeSort

```
10   typedef unsigned int ValueType;
11   typedef unsigned int* UnsignedVector;
```

```
66   void merge_sort_vector_internal (UnsignedVector vector, ValueType left, ValueType right) {
67       if(right > left) {
68           ValueType mid = left + (right - left) / 2;
69           merge_sort_vector_internal(vector, left, mid);
70           merge_sort_vector_internal(vector, mid + 1, right);
71           merge_vector(vector, left, mid, right);
72       }
73   }
74
75   void merge_sort_vector(UnsignedVector vector, ValueType left, ValueType right) {
76       --right;
77       merge_sort_vector_internal(vector, left, right);
78   }
```

A primeira função é onde de fato o mergesort é realizado, onde as chamadas recursivas acontecem, assim como a função de merge das metades ordenadas também é chamada.
A função de baixo é apenas uma interface para chamar a função de verdade, nela é feito apenas um decremento no valor da posição final, pois na versão paralela é feito mais que um decremento.

Algoritmo Serial do MergeSort

```
36  void merge_vector(UnsignedVector vector, ValueType left, ValueType mid, ValueType right) {
37      ValueType size_A = right - left + 1;
38      ValueType size_L = mid - left + 1;
39      ValueType size_R = size_A - size_L;
40
41      ValueType* L_aux = malloc(size_L*sizeof(ValueType));
42      ValueType* R_aux = malloc(size_R*sizeof(ValueType));
43
44      memcpy(L_aux, vector + left     , sizeof(ValueType)*size_L);
45      memcpy(R_aux, vector + (mid + 1), sizeof(ValueType)*size_R);
46
47      ValueType i = 0;
48      ValueType j = 0;
49      ValueType k = left;
50
51      while(i < size_L && j < size_R) {
52          vector[k++] = (L_aux[i] < R_aux[j]) ? L_aux[i++] : R_aux[j++];
53      }
54
55      if(i < size_L) {
56          memcpy(vector + k, L_aux + i, sizeof(ValueType)*(size_L-i));
57      }
58      else {
59          memcpy(vector + k, R_aux + j, sizeof(ValueType)*(size_R-j));
60      }
61
62      free(L_aux);
63      free(R_aux);
64  }
```

Ao lado temos a função onde o merge das metades ordenadas é feito.
São calculados os tamanhos dos vetores auxiliares, a memória auxiliar é alocada e os elementos são copiados.
No laço *while* é que os elementos são corretamente posicionados no vetor principal, sendo que de acordo com o elemento posicionados o devido índice é incrementado.
Ao fim do laço, os elementos restantes são copiados para as posições restantes.
Por fim, a memória é liberada.

Algoritmo Paralelo do MergeSort

```
118  void merge_sort_vector(UnsignedVector vector, ValueType left, ValueType right) {
119      ValueType distance = (right - left) / threads_number;
120      UnsignedVector limits = malloc((threads_number+1)*sizeof(double));
121
122      for (ValueType i = 0; i < threads_number; ++i) {
123          limits[i] = (left + i*distance);
124      }
125      limits[threads_number] = right;
126
127      #pragma omp parallel num_threads(threads_number) default(none) shared(vector, limits, threads_number)
128      {
129          #pragma omp for schedule(guided)
130          for (ValueType j = 0; j < threads_number; ++j) {
131              merge_sort_vector_internal(vector, limits[j], limits[j+1]-1);
132          }
133      }
134      merge_vectors(vector, limits, threads_number);
135
136      free(limits);
137  }
```

Essa é a interface da versão paralela do MergeSort, são calculados os limites que cada thread deve ordenar e por fim é feito o merge de todas as partes.

Algoritmo Paralelo do MergeSort

```
87  void merge_vectors(UnsignedVector vector, UnsignedVector limits, ValueType size) {
88      if(size == 2){
89          merge_vector(vector, limits[0], limits[1]-1, limits[2]-1);
90          return;
91      }
92      if(size == 3){
93          merge_vector(vector, limits[0], limits[1]-1, limits[2]-1);
94          merge_vector(vector, limits[0], limits[2]-1, limits[3]-1);
95          return;
96      }
97
98      #pragma omp parallel sections num_threads(threads_number) default(none) shared(vector, limits, size)
99      {
100         #pragma omp section
101         {
102             merge_vectors(vector, limits, (size/2) );
103         }
104         #pragma omp section
105         {
106             merge_vectors(vector, limits + (size/2), (size/2));
107         }
108     }
109
110     if(size % 2 == 0) {
111         merge_vector(vector, limits[0], limits[(size/2)]-1, limits[size]-1);
112     } else {
113         merge_vector(vector, limits[0], limits[(size/2)]-1, limits[size-1]-1);
114         merge_vector(vector, limits[0], limits[size-1]-1, limits[size]-1);
115     }
116 }
```

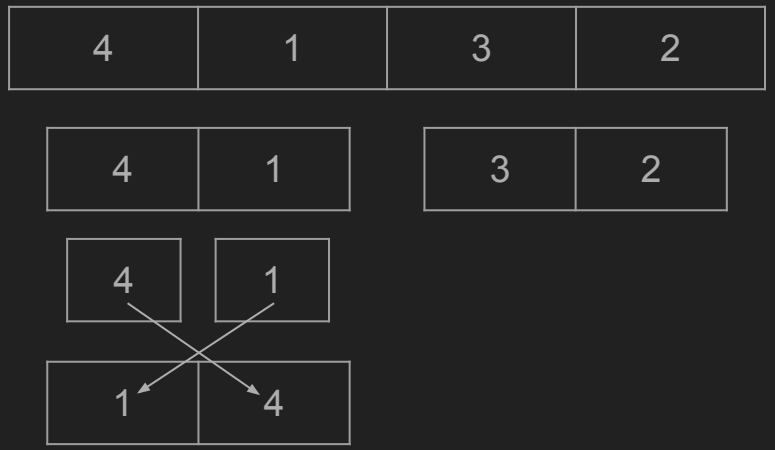
Na função de merge (*merge_vectors*) é novamente aplicada a estratégia de Dividir e Conquistar para recursivamente realizar o merge de todas as partes ordenadas por cada thread. Os casos bases são quando temos 2 ou 3 partes, Quando temos mais que 2 ou 3 partes é feita a chamada recursiva com uma thread responsável por cada metade.

Corretude do Algoritmo Serial do MergeSort

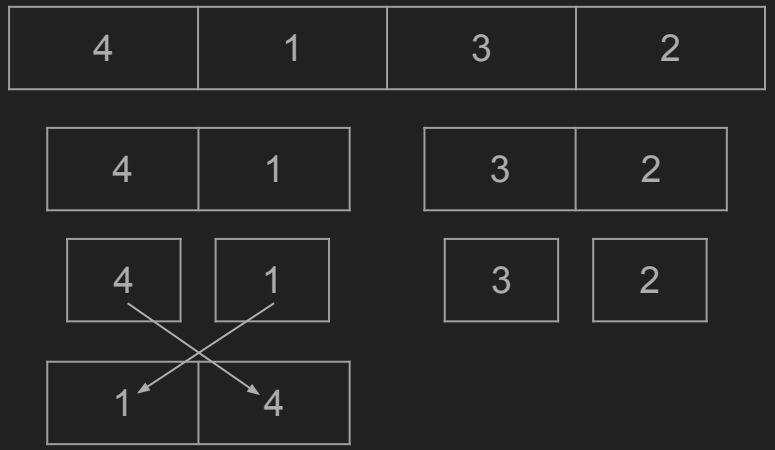
Corretude do Algoritmo Serial do MergeSort

Corretude do Algoritmo Serial do MergeSort

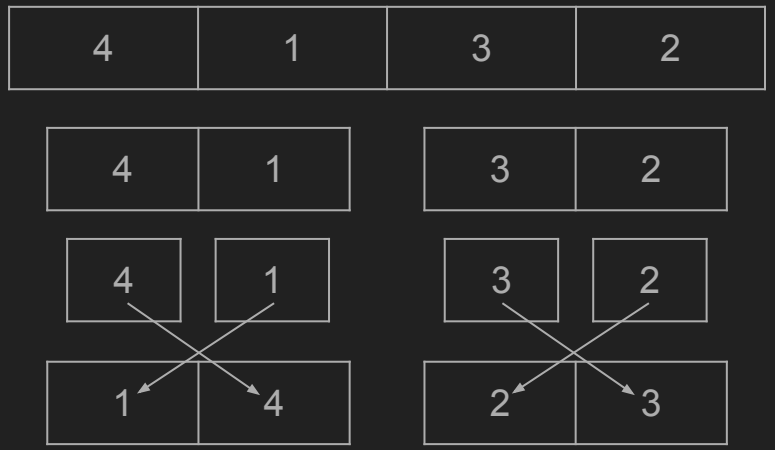
Corretude do Algoritmo Serial do MergeSort



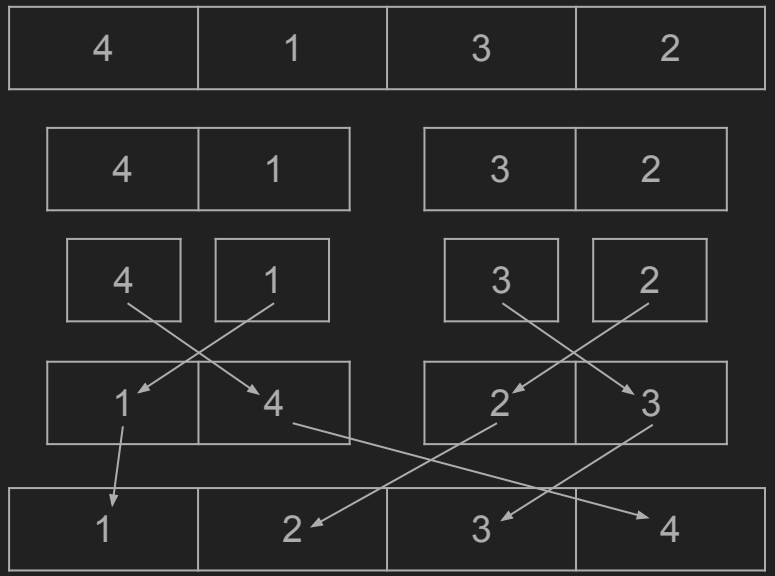
Corretude do Algoritmo Serial do MergeSort



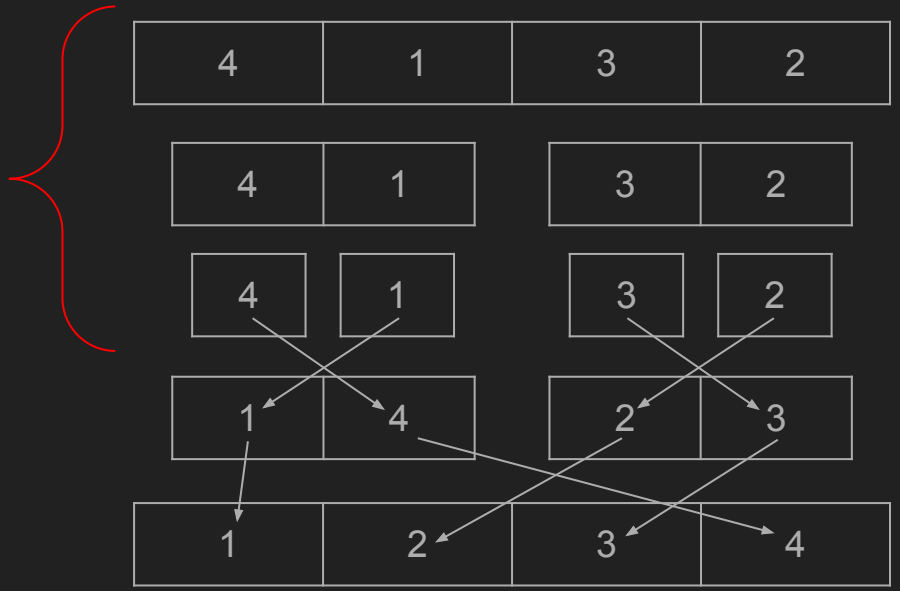
Corretude do Algoritmo Serial do MergeSort



Corretude do Algoritmo Serial do MergeSort

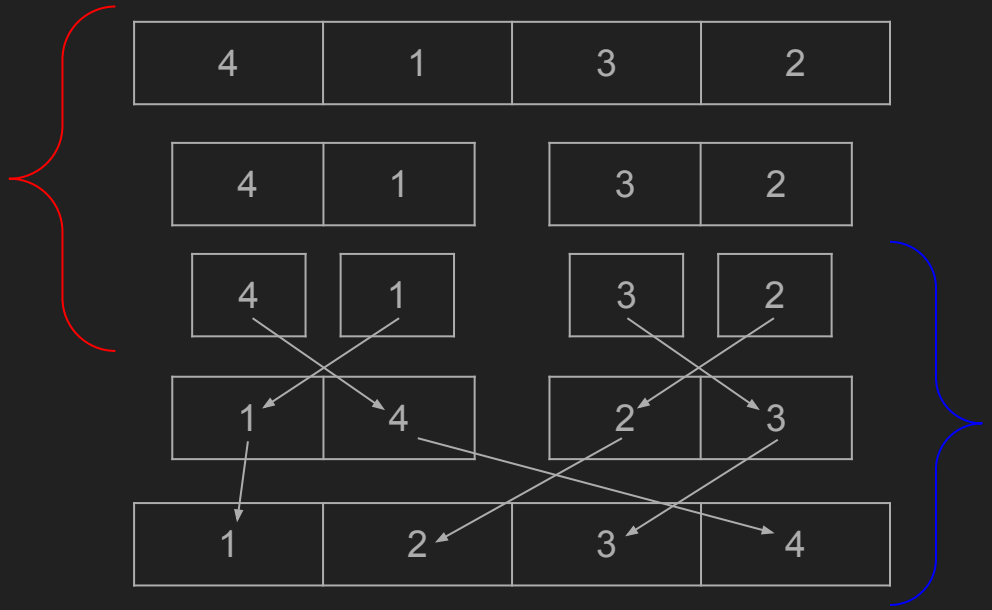


Corretude do Algoritmo Serial do MergeSort



Nas primeiras etapas temos a divisão em problemas menores (Divisão).

Corretude do Algoritmo Serial do MergeSort



Nas primeiras etapas temos a divisão em problemas menores (Divisão).
Nas etapas finais temos a resolução e combinação das etapas (Conquista).

Corretude do Algoritmo Paralelo do MergeSort

4	1	3	2

Com 2 threads

Corretude do Algoritmo Paralelo do MergeSort

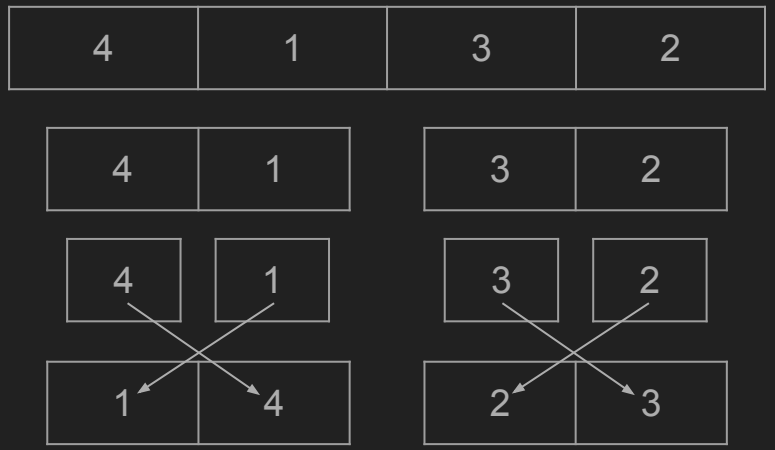
Com 2 threads

Corretude do Algoritmo Paralelo do MergeSort

Com 2 threads

A divisão da segunda parte ocorre em paralelo com a da primeira parte

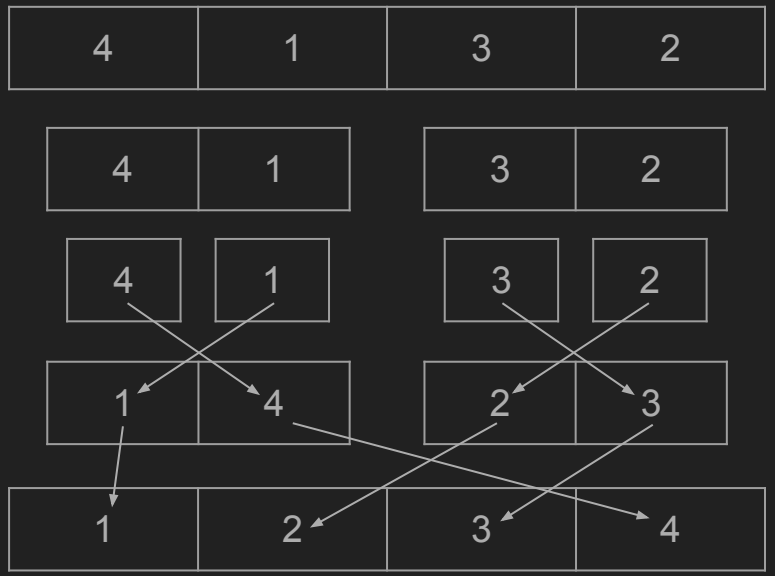
Corretude do Algoritmo Paralelo do MergeSort



Com 2 threads

A divisão da segunda parte ocorre em paralelo com a da primeira parte.
Assim como o merge interno é realizado em paralelo.

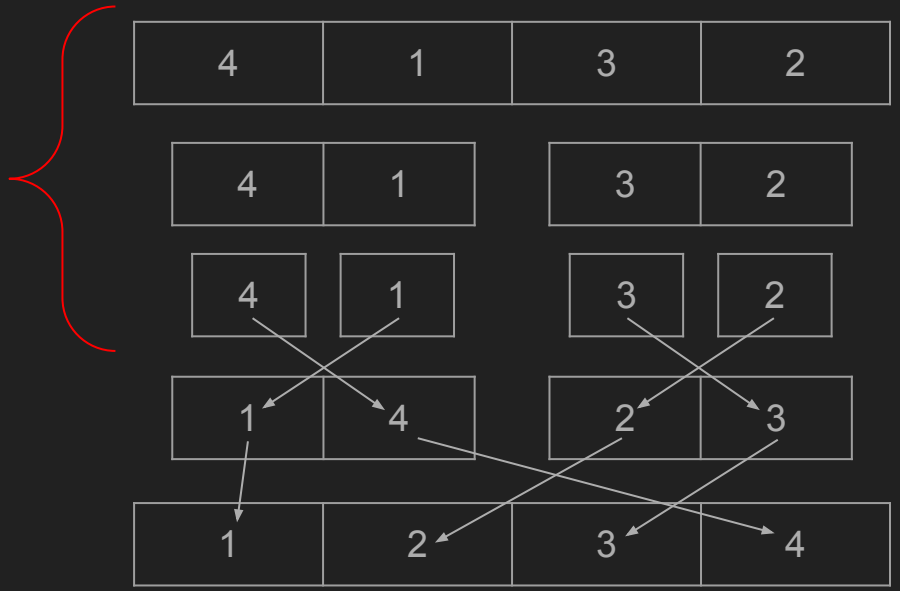
Corretude do Algoritmo Paralelo do MergeSort



Com 2 threads

A divisão da segunda parte ocorre em paralelo com a da primeira parte.
Assim como o merge interno é realizado em paralelo.
Com o término do ordenamento de cada parte, é realizado o merge das partes ordenadas. Obtendo assim o vetor final.

Corretude do Algoritmo Paralelo do MergeSort

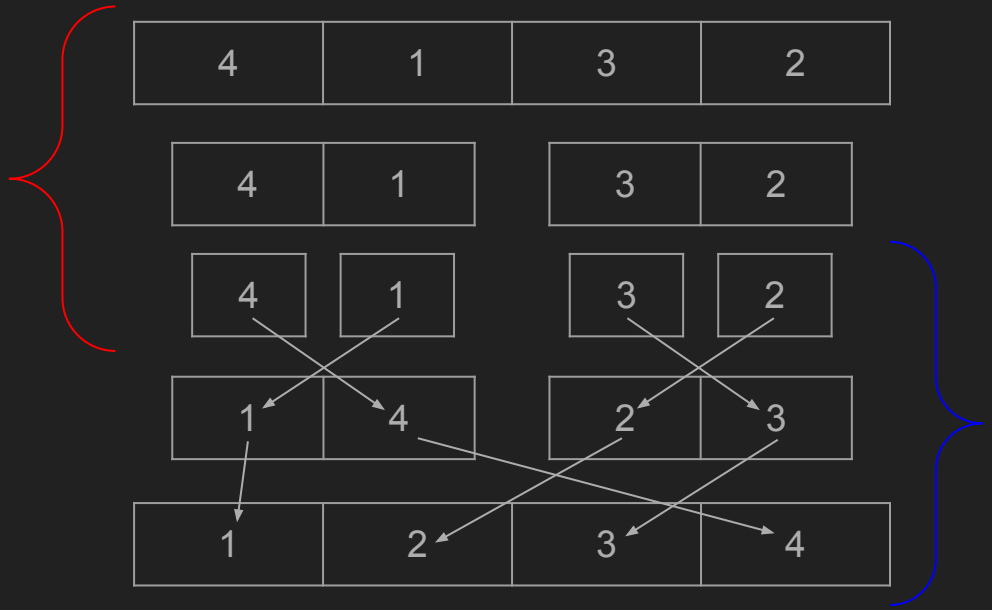


Com 2 threads:

A divisão da segunda parte ocorre em paralelo com a da primeira parte.
Assim como o merge interno é realizado em paralelo.
Com o término do ordenamento de cada parte, é realizado o merge das partes ordenadas. Obtendo assim o vetor final.

Novamente temos as etapas de:
Divisão

Corretude do Algoritmo Paralelo do MergeSort



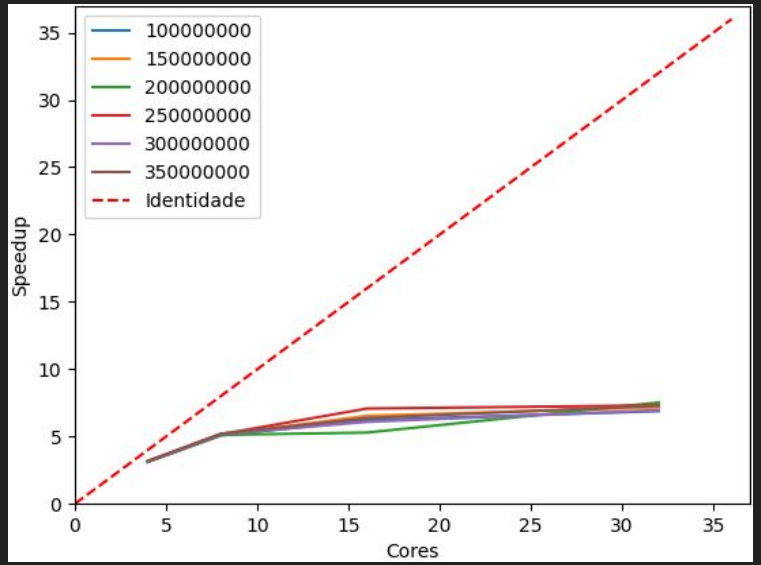
Com 2 threads:

A divisão da segunda parte ocorre em paralelo com a da primeira parte.
Assim como o merge interno é realizado em paralelo.
Com o término do ordenamento de cada parte, é realizado o merge das partes ordenadas. Obtendo assim o vetor final.

Novamente temos as etapas de:
Divisão e Conquista.

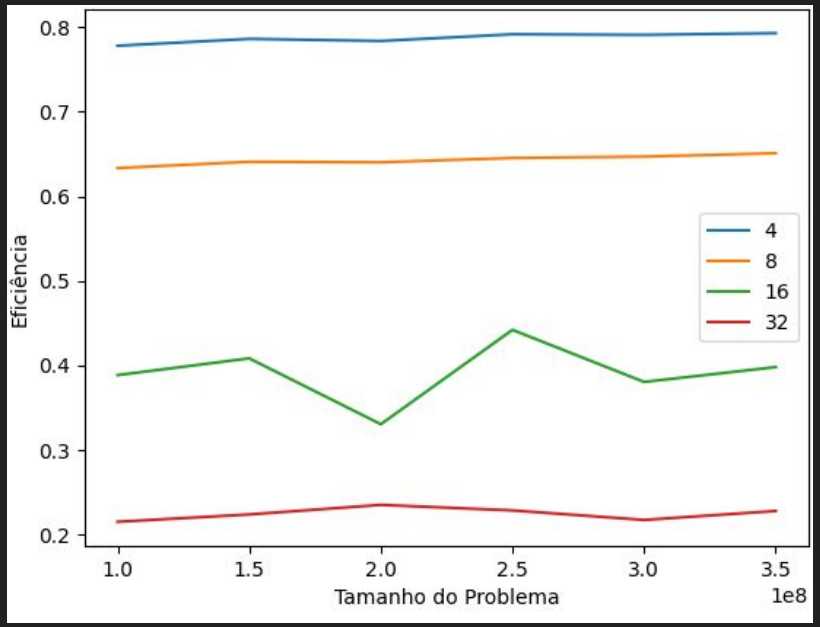
Análise de Speedup, Eficiência e Escalabilidade

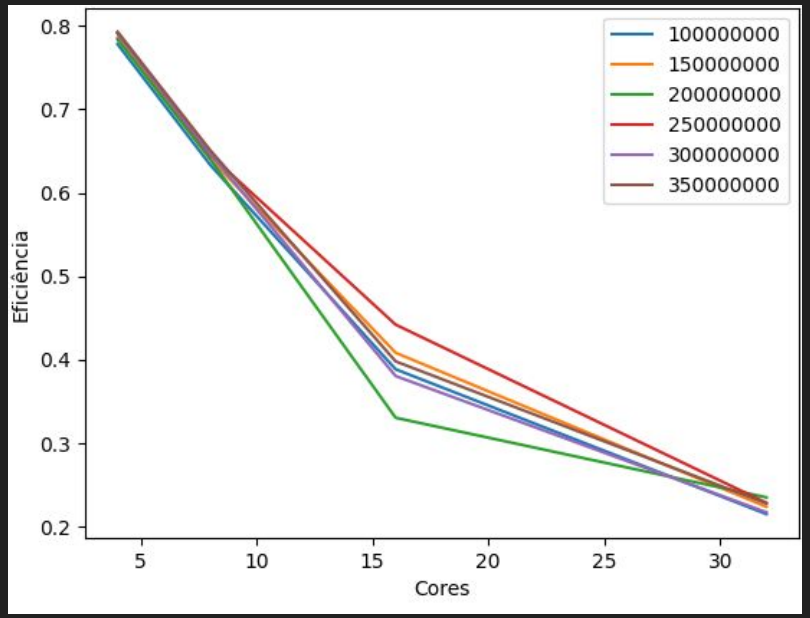
Tamanho do Problema	Serial	4	8	16	32
100000000	36,29818182	11,66203213	7,16241470	5,83763487	5,27261145
150000000	55,96000000	17,79348821	10,91436425	8,56394576	7,81364857
200000000	74,89909091	23,89023680	14,62228583	14,16228459	9,95496954
250000000	94,95818182	29,98619589	18,39116566	13,42333262	12,98262784
300000000	115,28272727	36,43567126	22,27351928	18,93479857	16,57769051
350000000	135,58272727	42,74521834	26,03315648	21,28726016	18,59280132



Perceptível que pelos tempos e pelo gráfico do speedup que o ganho de tempo com a paralelização não aumenta muito conforme o incremento do número de cores, alcançando uma assíntota relativamente rápido.

Análise de Speedup, Eficiência e Escalabilidade





Com relação aos gráficos de eficiência, por tamanho do problema e por número de cores. Vemos comportamentos comuns, A eficiência se mantém constante, com algumas flutuações de acordo com o tamanho do problema ao usar 16 cores, e diminui conforme aumentamos a quantidade de cores.

Análise de Speedup, Eficiência e Escalabilidade

Provavelmente esse é o limite que essa abordagem de paralelização pode alcançar, talvez utilizando outras técnicas sejam obtidos resultados melhores.

A versão do mergesort implementada pode ser classificada como **Fracamente Escalável**, visto que com o aumento do tamanho do problema e do número de cores em proporções iguais, a eficiência se mantém constante.

Considerações Finais

Assim, dado o que foi aqui apresentado, podemos concluir que:

- A ordenação de números é um problema conhecido na computação;
- O MergeSort é um dos algoritmos eficientes para a tarefa de ordenação;
- A paralelização do MergeSort utilizada possui um limite de speedup;
- Mesmo com esse limite, a eficiência possui um comportamento comum;
  - Tanto na variação de tamanho do problema para uma mesma quantidade de cores;
  - Como na variação da quantidade de cores para um mesmo tamanho de problema;
- A versão implementada do MergeSort paralelo se categoriza como:

Fracamente Escalável