UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
INSTITUTO METRÓPOLE DIGITAL
TÓPICOS ESPECIAIS EM INTERNET DAS COISAS “B” - T01
(2020.6)

Atividade 02 - Odd Even Transposition Sort

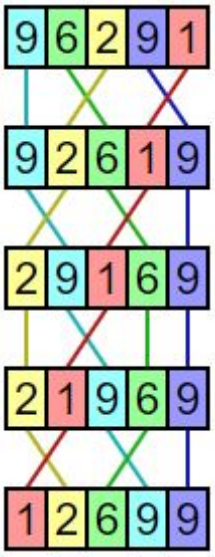
Paulo Eneas Rolim Bezerra

14 de outubro de 2020

1 INTRODUÇÃO

O problema proposto é fornecer ao programa uma série de números inteiros e ordená-los na ordem crescente de valor..

A idéia para solucionar o problema foi utilizar o método da classificação ímpar-par (*odd-even sort*) iterando a lista de números inicial, comparando os elementos adjacentes e trocando-os de posição se estiverem na ordem errada.



A característica única da classificação ímpar-par, se deve a forma como as iterações da classificação alternam entre classificar pares indexados ímpar / par e par / ímpar.

2. DESENVOLVIMENTO

2.1. Solução Serial Implementada em C++ para a ordenação ímpar-par

A solução serial foi implementada na linguagem de programação C++, e conforme pode ser visto na figura abaixo, foi criada uma função denominada “ordenação”, que recebe como entrada um vetor de inteiro e na saída da função os elementos do vetor entrada são ordenados do menor para o maior.

```
8   int ordenacao(int vetor[tamanho_problema]){
9
10      int var_local;
11      int estagio;
12      int i;
13
14      for (estagio = 0; estagio < tamanho_problema; estagio++){
15          if (estagio % 2 == 0){ // Fase par da ordenação
16
17              for (i = 1; i < tamanho_problema; i+=2){
18
19                  if (vetor[i-1] > vetor[i]){
20
21                      var_local = vetor[i];
22                      vetor[i] = vetor[i - 1];
23                      vetor[i - 1] = var_local;
24                  }
25              }
26          }else{ // Fase ímpar da ordenação
27
28              for (i = 1; i < tamanho_problema - 1; i+=2){
29
30                  if (vetor[i] > vetor[i+1]){
31
32                      var_local = vetor[i];
33                      vetor[i] = vetor[i+1];
34                      vetor[i+1] = var_local;
35                  }
36              }
37          }
38      }
39
40      return vetor[tamanho_problema];
```

Em seguida na função principal, foi criado um vetor de inteiros que é inicializado por um laço de repetição, nesse laço são gerados número inteiros de 0 à 999 e inseridos no vetor entrada. O tamanho do vetor e a quantidade de elementos inseridos é igual ao tamanho do problema. Vejamos:

```
44  int main(int argc, char *argv[]){
45
46      struct timeval start, stop;
47
48      int vetor_entrada[tamanho_problema];
49
50      //std::cout << "Númermos que serão ordenados: " << std::endl;
51
52      gettimeofday(&start, 0);
53
54      for (int i = 0; i < tamanho_problema; i++){
55
56          std::random_device rd;
57          std::default_random_engine gen(rd());
58          std::uniform_int_distribution<>dis(0,999);
59          int valor_aleatorio = std::round(dis(gen));
60
61          vetor_entrada[i] = valor_aleatorio;
62         // std::cout << vetor_entrada[i] << " ";
63
64      }
```

Em seguida a função “ordenação” é chamada e o vetor entrada, já inicializado, é passado como parâmetro. Após ordenado o programa faz o registro do tempo de execução num arquivo de extensão .TXT e é encerrado.

```
68      ordenacao(vetor_entrada);
69      /*
70      for (int o = 0; o < tamanho_problema; o++)
71      {
72          std::cout << vetor_entrada[o] << " ";
73      }
74      std::cout << std::endl;
75      */
76      gettimeofday(&stop, 0);
77
78      FILE *fp;
79          char outputFilename[] = "tempo_de_exe_serial.txt";
80
81          fp = fopen(outputFilename, "a");
82          if (fp == NULL) {
83              fprintf(stderr, "Can't open output file %s!\n", outputFilename);
84              exit(1);
85          }
86
87          //testes de impressão no arquivo
88          fprintf(fp, "\t%1.2e ", (double)(stop.tv_usec - start.tv_usec) / 1000000 + (double)(stop.tv_sec - start.tv_s
89          fprintf(fp, "\t%d ",tamanho_problema); //tamanho do problema+1
90
91          fclose(fp);
92
93      return 0;
94  }
95
```

2.1.1. Solução Paralela Implementada em C++ para a ordenação ímpar-par

Seguindo a mesma lógica aplicada no programa serial, onde um vetor de inteiros é declarado e inicializado por um laço de repetição, o programa paralelo se difere ao utiliza a função “MPI_Scatter” para dividir o vetor de entrada em partes proporcionais ao número de processos selecionados para a execução do programa. Vejamos abaixo:

```
38      tamanho_msg = tamanho_problema / comm_sz; /*variável responsável por armazenar o tamanho
39                                                  do vetor que será enviado para cada processo*/
40      int vetor_local[tamanho_msg];
41      /*
42      if (my_rank == 0)
43      {
44          std::cout << "Vetor INICIAL desordenado: ";
45          for (int i = 0; i < tamanho_problema; i++)
46          {
47              std::cout << vetor_entrada[i] << " ";   //Imprime na tela o vetor entrada
48          }
49          std::cout << " " << std::endl;
50
51
52      }*/
53      // separa o vetor entrada em partes e envia para cada processo
54      gettimeofday(&start, 0);
55
56      MPI_Scatter (vetor_entrada, tamanho_msg, MPI_INT, &vetor_local, tamanho_msg, MPI_INT, 0, MPI_COMM_WORLD);
```

Cada processo recebe uma secção do vetor entrada e realiza a ordenação dessa fatia, utilizando a lógica da ordenação ímpar-par (*odd-even sort*), e após todos os processos terminarem a ordenação de sua parte, o processo mestre recebe os pedaços de vetor, semi-ordenados, por intermédio da função “MPI_Gatter” que junta os pedaços do vetor entrada e o envia para o processo mestre. Além disso foi utilizada a função MPI_Barrier, para sincronizar o envio da mensagem e garantir que todos as secções do vetor entrada sejam enviadas ao mesmo tempo para o processo mestre.

```
105     MPI_Gather(vetor_local,tamanho_msg,MPI_INT,vetor_entrada,tamanho_msg,MPI_INT,0,MPI_COMM_WORLD);
106
107     MPI_Barrier(MPI_COMM_WORLD); //sincroniza as mensagens antes de enviar
```

O final o processo mestre ordena novamente o vetor e o tempo de execução do programa é registrado num de extensão .TXT e é encerrado.

3. RESULTADOS: Speedup, Eficiência e Escalabilidade

Foram realizados testes com os códigos serial e paralelo, com tamanhos de problema diferentes e o tempo de execução medido, conforme tabelas abaixo:

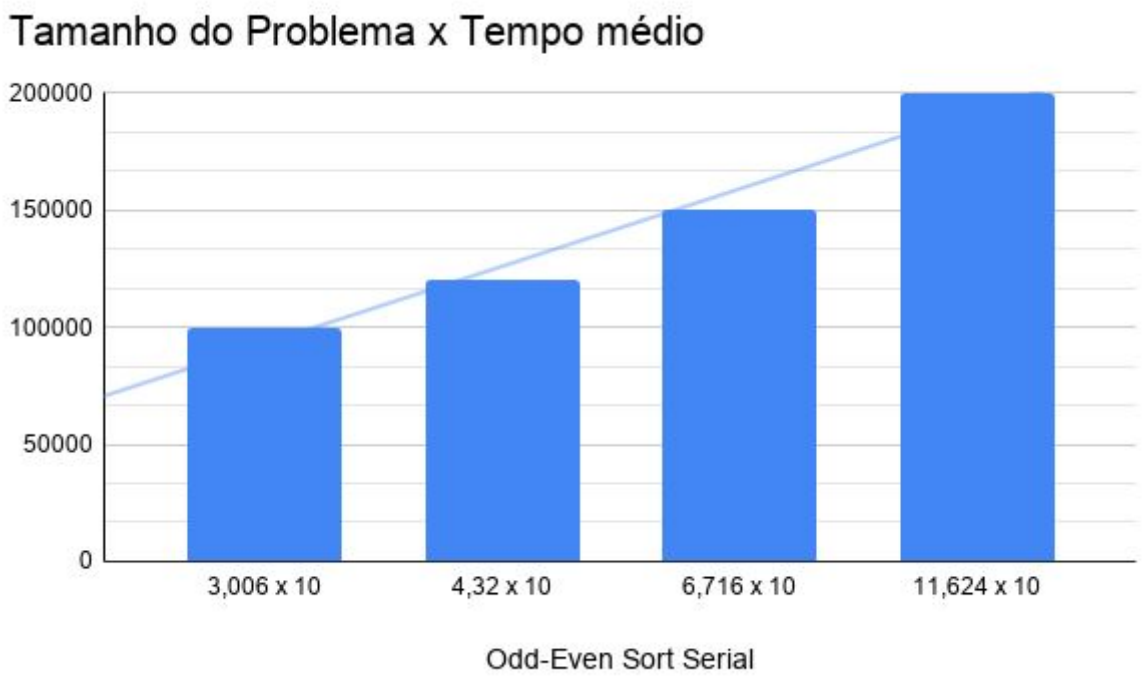
Tabela de Tempo de Execução - Odd-Even Serial		
	Tamanho do Problema	Tempo médio
Execução 01	100000	3,006 x 10
Execução 02	120000	4,320 x 10
Execução 03	150000	6,716 x 10
Execução 04	200000	11,624 x 10

Tabela de Tempo de Execução - Odd-Even Paralelo			
	Tamanho do Problema	Cores	Tempo médio
Execução 01	100000	2	2,708 x 10
Execução 02	120000	2	3,890 x 10
Execução 03	150000	2	6,036 x 10
Execução 04	200000	2	10,84 x 10

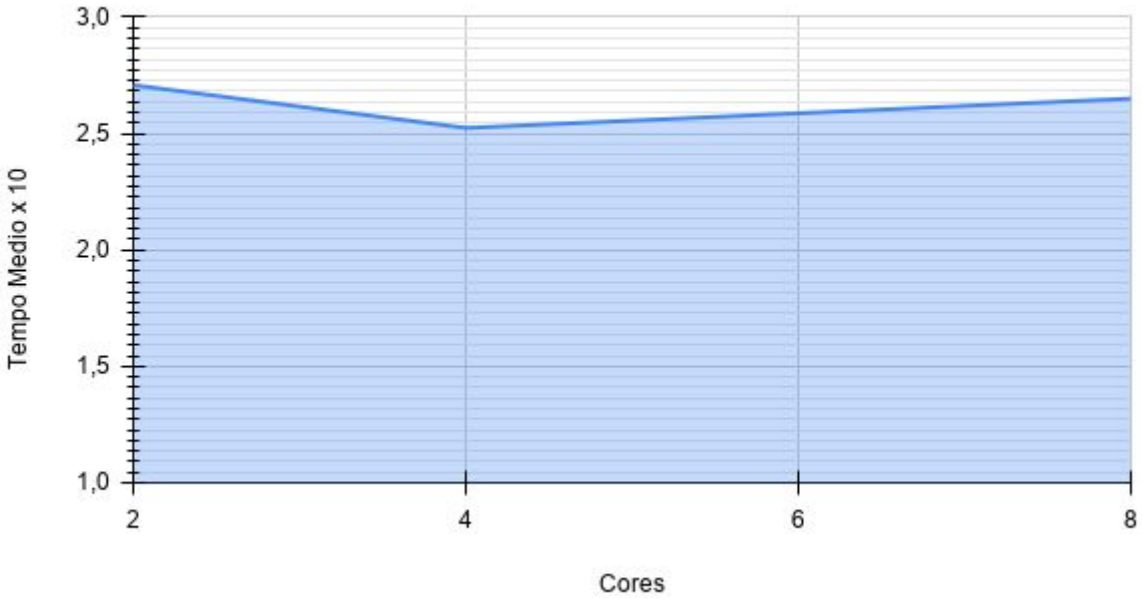
Tabela de Tempo de Execução - Odd-Even Paralelo			
	Tamanho do Problema	Cores	Tempo médio
Execução 01	100000	4	2,524 x 10
Execução 02	120000	4	3,606 x 10
Execução 03	150000	4	5,708 x 10
Execução 04	200000	4	10,154 x 10

Tabela de Tempo de Execução - Odd-Even Paralelo			
	Tamanho do Problema	Cores	Tempo médio
Execução 01	100000	8	2,650 x 10
Execução 02	120000	8	3,756 x 10
Execução 03	150000	8	6,108 x 10
Execução 04	200000	8	10,76 x 10

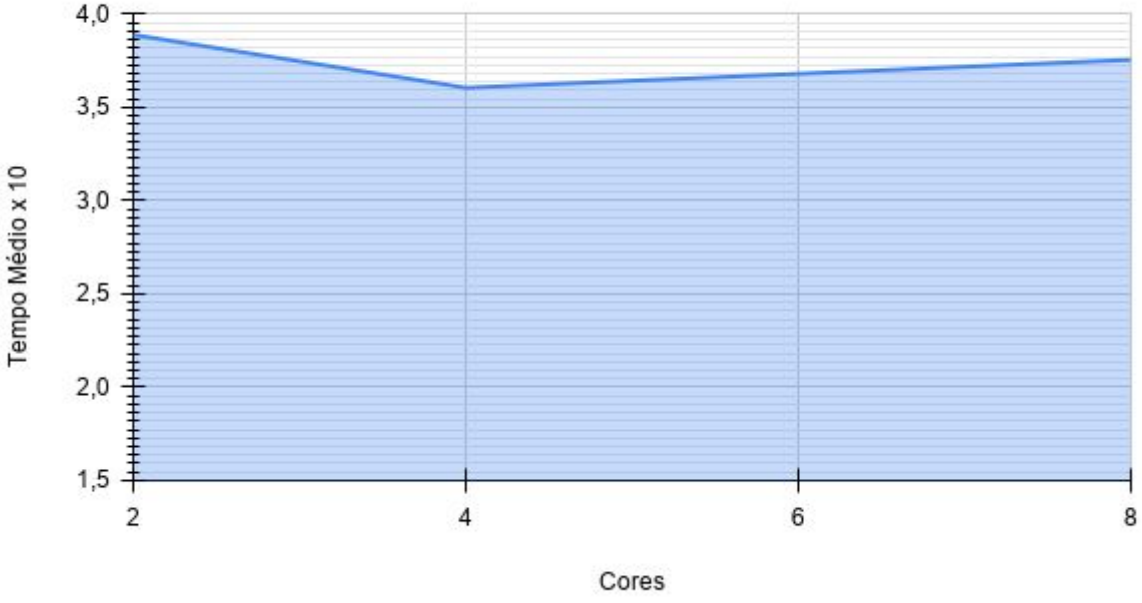
O computador de testes roda o Sistema Operacional Ubuntu 20.04.1 LTS, usa a placa mãe da fabricante Gigabyte modelo 970A-DS3P com o Base Clock (BCLK) setado em 201,51 Mhz, o processador é AMD-FX8320e de 8 núcleos e 16mb de memória cache com overclock para 3.5 Ghz, 16gb memória ram PC3-10700H DR3 O.C. 1600 MHz da fabricante Corsair com as seguintes latências principais 9, 9, 9, 24, 41.



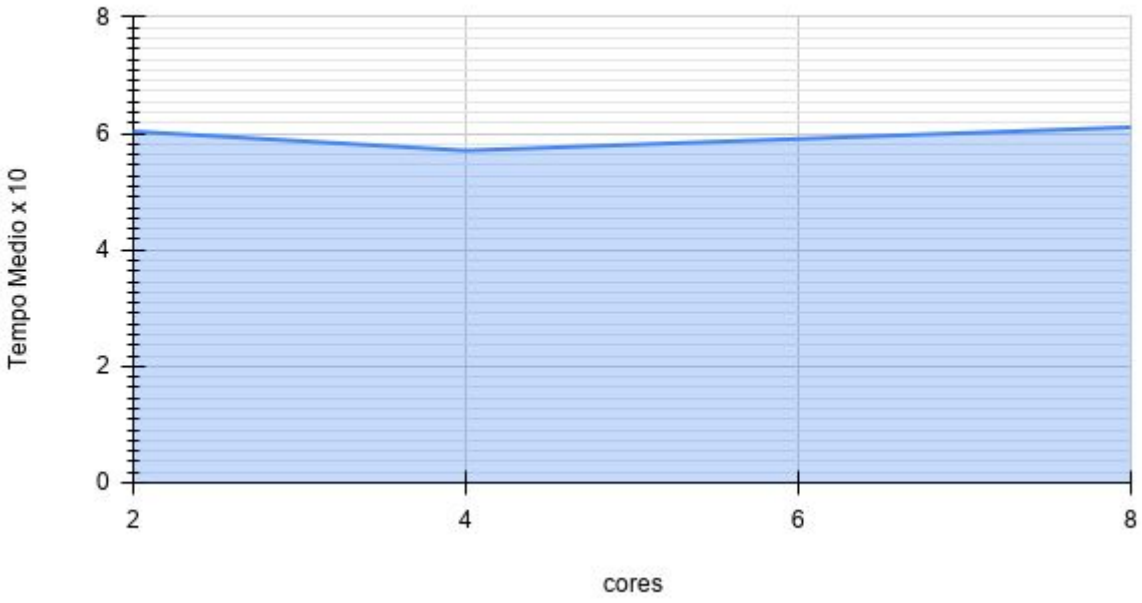
Tempo Medio x Cores - Tamanho Problema 100000



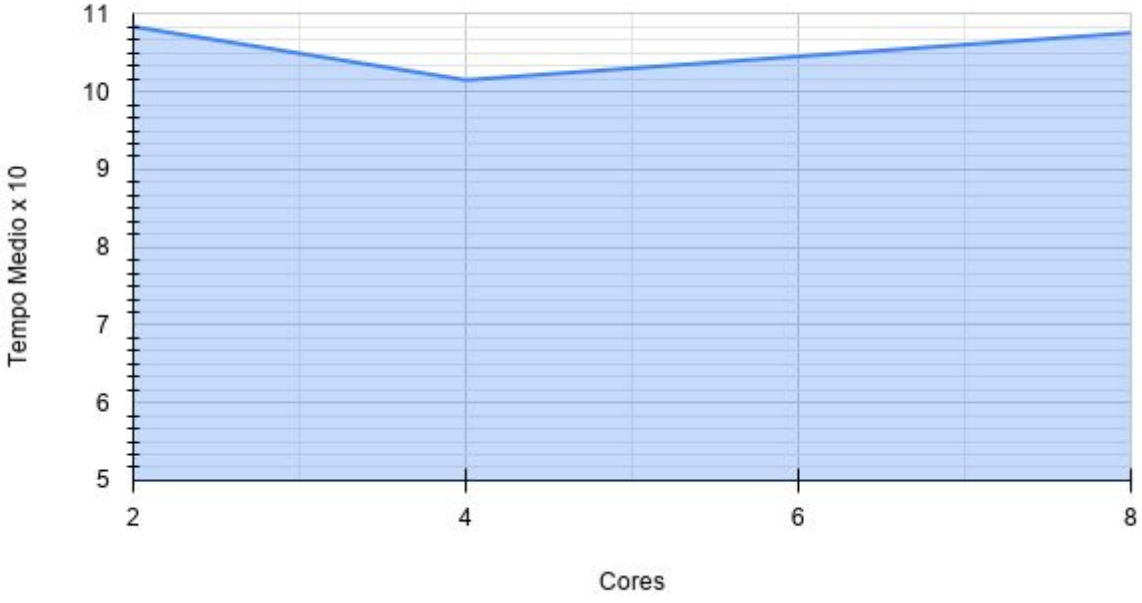
Tempo Médio x Cores - Tamanho Problema 120000

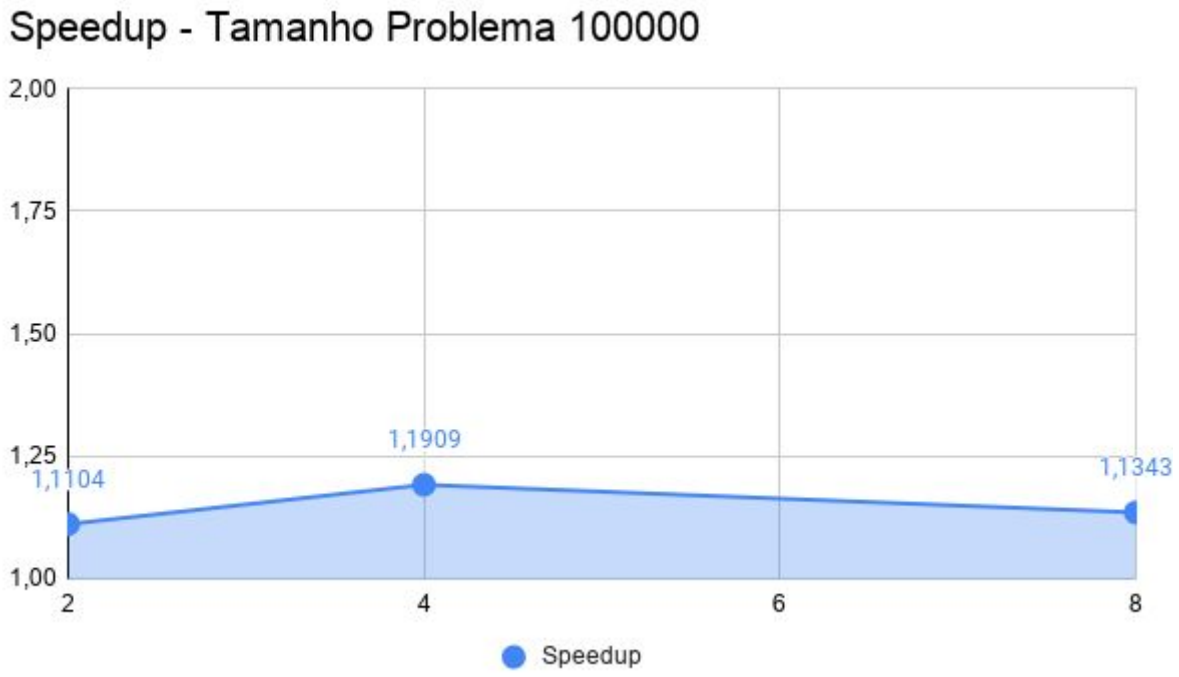


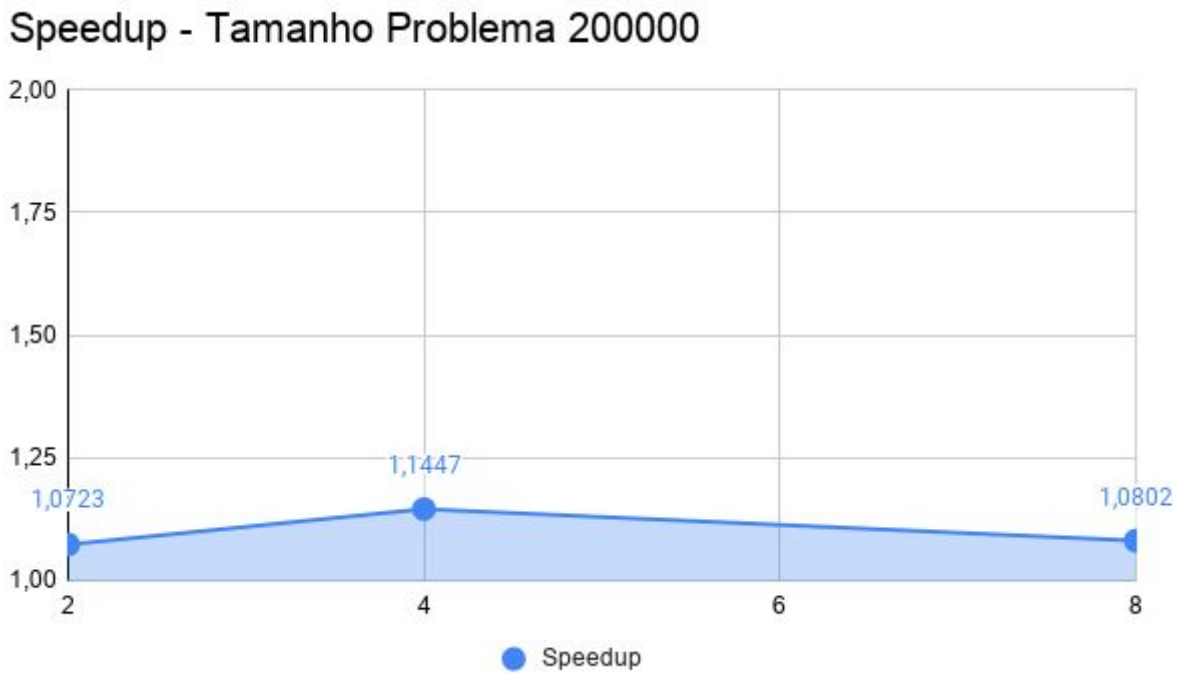
Tempo Medio x Cores - Tamanho Problema 150000



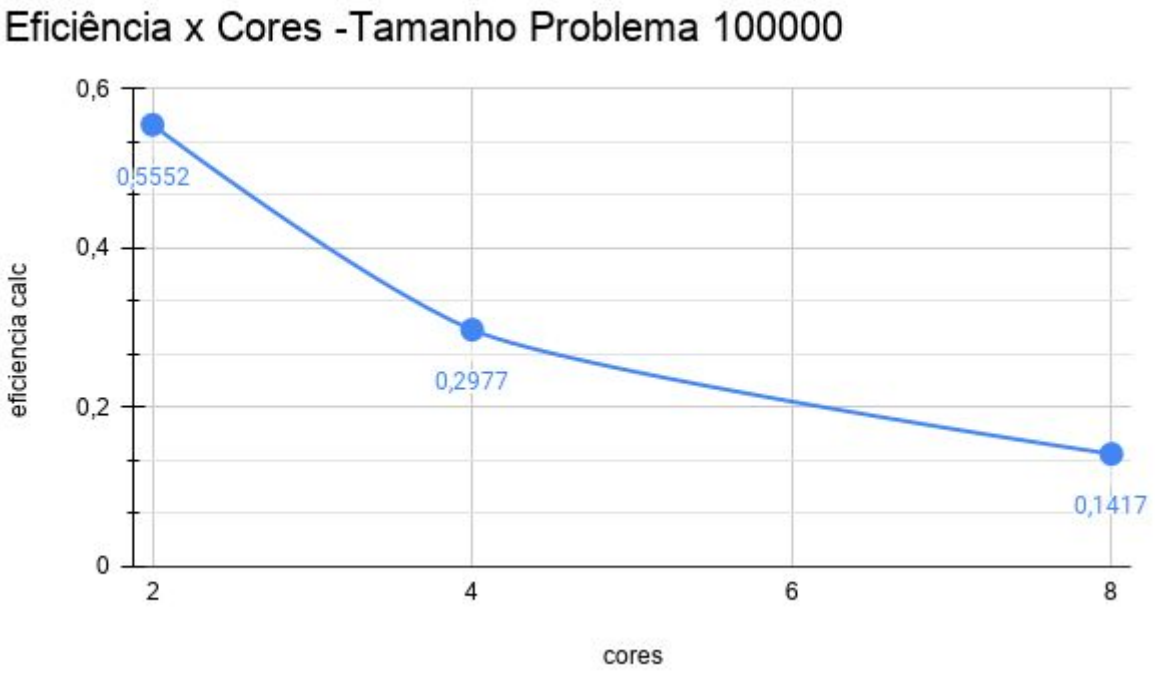
Tempo Medio x Cores - Tamanho Problema 200000

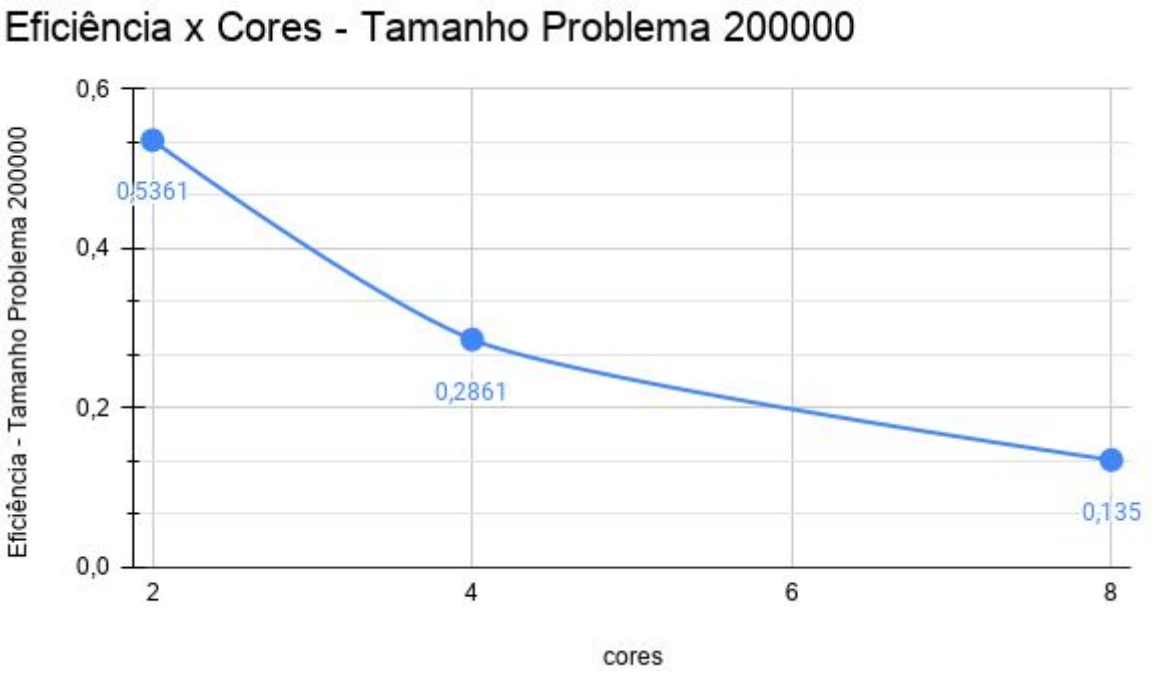






O *speedup* verificado nos testes não foi muito grande, pois no algoritmo produzido há grande dependência entres os cores, além disso dada a arquitetura do processador utilizado nos testes, que possui quatro cores completos e quatro ULA's e não possui "*Hyper-threading*" o melhor *speedup* foi aferido ao rodar o programa paralelo com 4 cores.





Os gráficos acima demonstram a eficiência do código paralelo em relação ao número de cores, e pela análise da eficiência constatou-se que o algoritmo é fracamente escalável, pois apesar do aumento de speedup com o aumento do número de cores, a eficiência não cresceu.