UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

MPI 2 - Odd-even Transposition Sort

Pedro Cardoso Carvalho

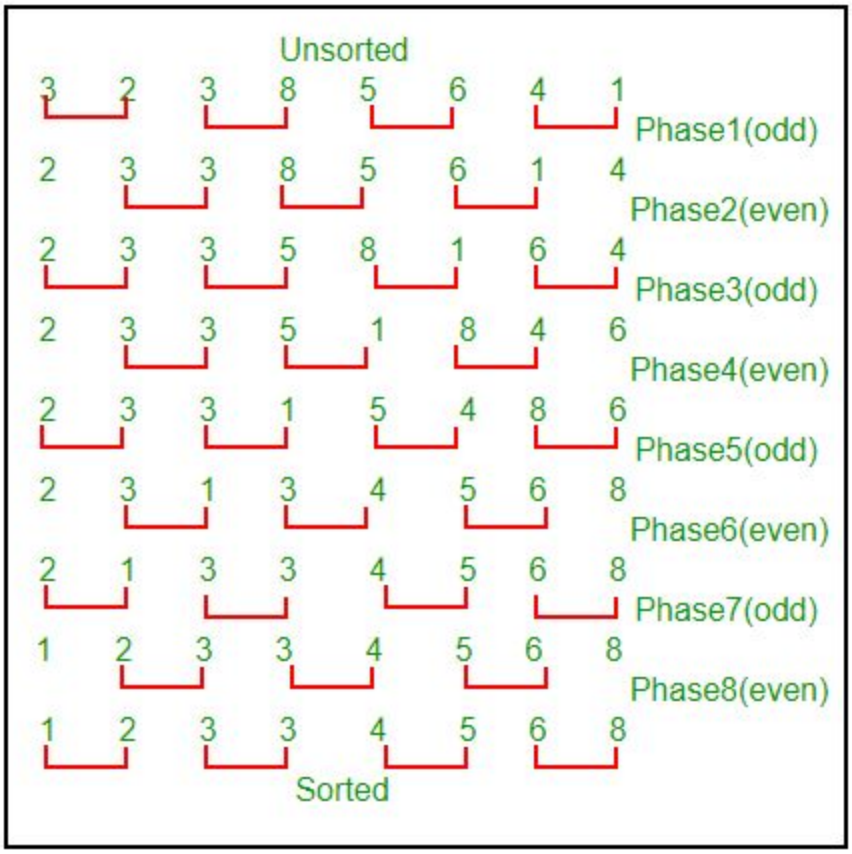
NATAL-RN
2020

Sumário

Introdução

Neste relatório iremos realizar um comparativo entre a utilização da programação paralela à tradicional programação serial no método de ordenação “Odd-even Transposition sort”.Esta forma de ordenação de lista parte de um looping de N repetições, sendo N o número de elementos da lista, onde divide em momentos impar e momentos pares do looping.Como demonstrado na figura 1.

Figura 1- Demonstração do Odd-even Transposition sort



Fonte -GeekForGeek

Nos momentos pares é criado pares de teste partindo do primeiro elemento da lista e caso seja identificado que o valor da direita é menor que o valor da esquerda invertemos a posição desses dois valores.Para as fases ímpar do looping temos a mesma geração de pares mas nesse caso inicia no segundo valor da lista. Ao final temos uma lista ordenado.

Desenvolvimento

Ambiente de teste

Todos os teste foram realizados em uma máquina com processador i5 8265u de frequência de 1.60-3.90GHz e 8Gb de memória RAM. Foram selecionados 4 tamanhos de problemas sendo eles uma lista de 128000, 140000, 150000 e 160000 elementos, para cada uma desses casos foram realizados 5 execuções e para o caso do código paralelo para cada ambiente deste temos os casos de utilização de 2, 4 e 8 processos. Para geração dos valores da lista serem iguais para todos os casos foi utilizado a semente com valor fixo de 10 em hardcode.

Programa serial

Para código serial, seguindo a ideia do problema relatado na introdução, foi gerado duas sequência de looping sendo o primeiro o principal que identifica os momentos pares e ímpares e o segundo ,dependerá do momento, segue um looping que possui index sendo somado de dois em dois e realizado a comparação e inversão, caso necessário,dos pares(como mostrado no código da figura 2).

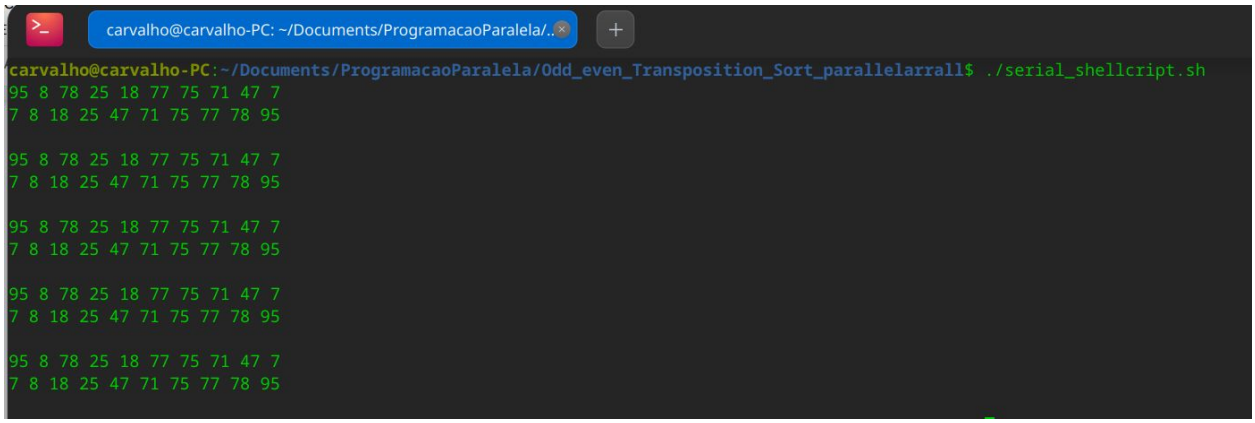
Figura 2- Codigo serial de Odd-even Transposition sort

```
10      int N = atoi(argv[1]);
11      srand(10);
12      int temp;
13      int array[N];
14      for (int i = 0; i < N; i++)
15      {
16          array[i] = rand()%100;
17      }
18      for (int phase = 0; phase < N; phase++)
19      {
20          if (phase%2==0){
21              for (int i = 1; i < N; i+=2)
22              {
23                  if (array[i-1]> array[i]){
24                      temp = array[i];
25                      array[i] = array[i-1];
26                      array[i-1]= temp;
27                  }
28              }
29
30          }
31          else
32          {
33              for (int i = 1; i < N-1; i+=2)
34              {
35                  if (array[i]>array[i+1]){
36                      temp = array[i];
37                      array[i] = array[i+1];
38                      array[i+1]= temp;
39                  }
40              }
41
42          }
43
44      }
```

Fonte -Autor

Um exemplo de execução do código mostrado pode ser visto na figura 3 com uma lista de tamanho 10 sendo executada 5 vezes.

Figura 3- Execução Codigo serial de Odd-even Transposition sort



Fonte -Autor

Programa Paralelo

Para um modelo paralelo foi pensando em uma estratégia onde realizamos a divisão da lista em quantidade igual para cada processo. Inicialmente esse processo irá realiza a ordenação local da lista de valores enviados a ele. Posteriormente iniciamos o looping principal de momentos ímpares e pares, mas neste caso invés de realizarmos essa sequência realizando a checagem por pares de valores na lista, os pares serão feito pelos processos, como se os ids fossem uma lista ordenada e cada par realizará uma organização da união das duas lista locais e separar novamente para cada lado. Essa estrategia pode ser vista na figura 4.

Figura 4- Estrategia do codigo paralelo para Odd-even Transposition sort

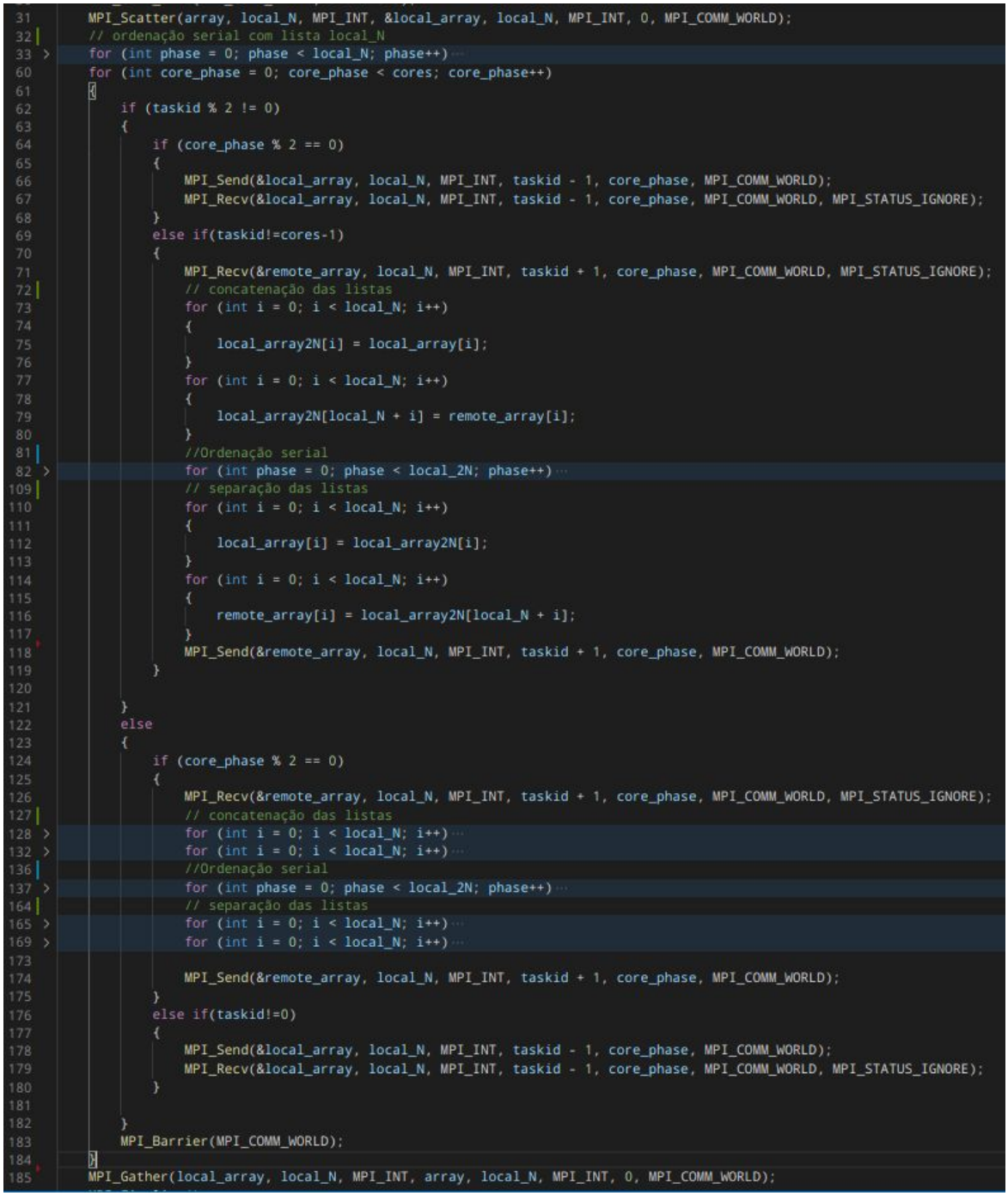
Time	Process			
	0	1	2	3
Start	15, 11, 9, 16	3, 14, 8, 7	4, 6, 12, 10	5, 2, 13, 1
After Local Sort	9, 11, 15, 16	3, 7, 8, 14	4, 6, 10, 12	1, 2, 5, 13
After Phase 0	3, 7, 8, 9	11, 14, 15, 16	1, 2, 4, 5	6, 10, 12, 13
After Phase 1	3, 7, 8, 9	1, 2, 4, 5	11, 14, 15, 16	6, 10, 12, 13
After Phase 2	1, 2, 3, 4	5, 7, 8, 9	6, 10, 11, 12	13, 14, 15, 16
After Phase 3	1, 2, 3, 4	5, 6, 7, 8	9, 10, 11, 12	13, 14, 15, 16

Fonte - Aula Capítulo 3 - Aula 8 - MPI - Odd even Transposition Sort

Para a ordenação entre dois processo foram idealizado um envio da lista local do processo de maior índice para o de menor índice e neste foi realizado uma ordenação igual o mostrado no código serial. Ao final dos processo temos a concatenação de todas as listas na ordem dos valores dos id de processos.

Para o código funcionar utilizamos as funções: MPI_Scatter(para a divisão da lista igualmente para cada processo), MPI_Gather(para união de todas as lista locais de processo por ordem de indices), MPI_Barrier(para forçar uma sincronização dos processo ao final de cada momento par ou ímpar) e par de MPI_Recv e MPI_Send (para comunicação entre os processos). Após o recebimento da lista de outro processo temos que utilizar looping para unir com a lista local em uma estrutura com o dobro do tamanho, para realizar a ordenação e no final um novo processo de separação e envio. Codigo pode ser visto na figura 5.

Figura 5- Código paralelo de Odd-even Transposition sort



```
31      MPI_Scatter(array, local_N, MPI_INT, &local_array, local_N, MPI_INT, 0, MPI_COMM_WORLD);
32      // ordenação serial com lista local_N
33  >   for (int phase = 0; phase < local_N; phase++)···
60      for (int core_phase = 0; core_phase < cores; core_phase++)
61      {
62          if (taskid % 2 != 0)
63          {
64              if (core_phase % 2 == 0)
65              {
66                  MPI_Send(&local_array, local_N, MPI_INT, taskid - 1, core_phase, MPI_COMM_WORLD);
67                  MPI_Recv(&local_array, local_N, MPI_INT, taskid - 1, core_phase, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
68              }
69              else if(taskid!=cores-1)
70              {
71                  MPI_Recv(&remote_array, local_N, MPI_INT, taskid + 1, core_phase, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
72                  // concatenação das listas
73                  for (int i = 0; i < local_N; i++)
74                  {
75                      local_array2N[i] = local_array[i];
76                  }
77                  for (int i = 0; i < local_N; i++)
78                  {
79                      local_array2N[local_N + i] = remote_array[i];
80                  }
81                  //Ordenação serial
82  >               for (int phase = 0; phase < local_2N; phase++)···
109                 // separação das listas
110                 for (int i = 0; i < local_N; i++)
111                 {
112                     local_array[i] = local_array2N[i];
113                 }
114                 for (int i = 0; i < local_N; i++)
115                 {
116                     remote_array[i] = local_array2N[local_N + i];
117                 }
118                 MPI_Send(&remote_array, local_N, MPI_INT, taskid + 1, core_phase, MPI_COMM_WORLD);
119             }
120
121         }
122         else
123         {
124             if (core_phase % 2 == 0)
125             {
126                 MPI_Recv(&remote_array, local_N, MPI_INT, taskid + 1, core_phase, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
127                 // concatenação das listas
128 >               for (int i = 0; i < local_N; i++)···
132 >               for (int i = 0; i < local_N; i++)···
136                 //Ordenação serial
137 >               for (int phase = 0; phase < local_2N; phase++)···
164                 // separação das listas
165 >               for (int i = 0; i < local_N; i++)···
169 >               for (int i = 0; i < local_N; i++)···
173
174                 MPI_Send(&remote_array, local_N, MPI_INT, taskid + 1, core_phase, MPI_COMM_WORLD);
175             }
176             else if(taskid!=0)
177             {
178                 MPI_Send(&local_array, local_N, MPI_INT, taskid - 1, core_phase, MPI_COMM_WORLD);
179                 MPI_Recv(&local_array, local_N, MPI_INT, taskid - 1, core_phase, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
180             }
181
182         }
183         MPI_Barrier(MPI_COMM_WORLD);
184     }
185     MPI_Gather(local_array, local_N, MPI_INT, array, local_N, MPI_INT, 0, MPI_COMM_WORLD);
```

Fonte -Autor

Um exemplo de execução do código pode ser visto na figura 6, com a utilização da ordenação de um array de 16 elemento em 2, 4 e 8 processos.

Figura 6- Execução Codigo paralelo de Odd-even Transposition sort

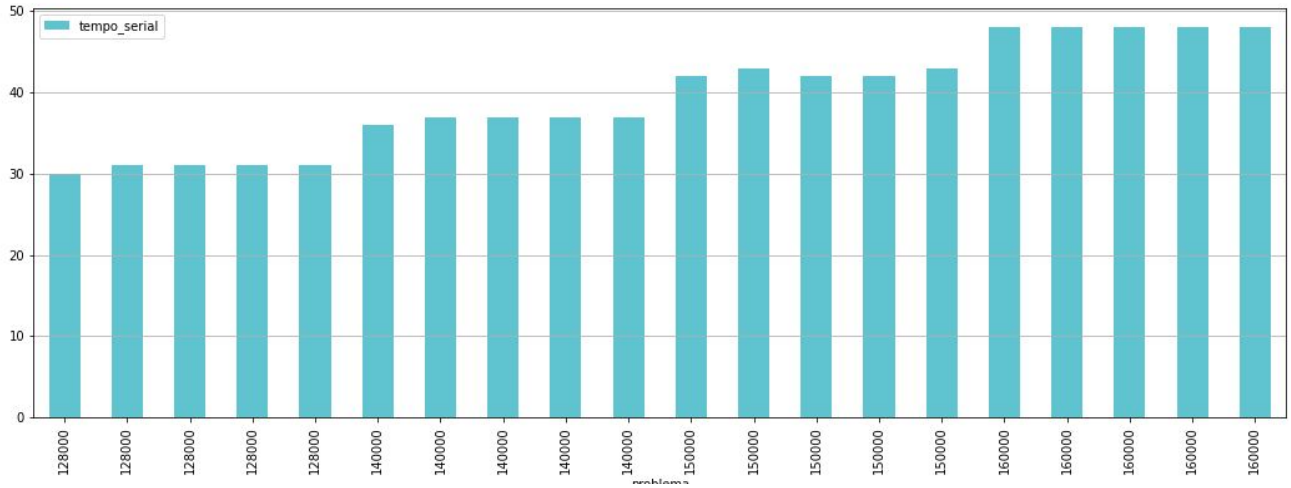
```
carvalho@carvalho-PC: ~/Documents/ProgramacaoParalela/..
carvalho@carvalho-PC:~/Documents/ProgramacaoParalela/Odd_even_Transposition_Sort_parallelarrall$ ./parallel_shellscript.sh
Cores:2
20 38 71 68 68 44 55 81 62 28 12 23 88 66 57 83
7 8 18 23 25 28 44 47 71 73 73 75 77 78 81 95
Cores:4
20 38 71 68 68 44 55 81 62 28 12 23 88 66 57 83
7 8 18 23 25 28 44 47 71 73 73 75 77 78 81 95
Cores:8
20 38 71 68 68 44 55 81 62 28 12 23 88 66 57 83
7 8 18 23 25 28 44 47 71 73 73 75 77 78 81 95
```

Fonte -Autor

Resultados

Executando o código serial em um ambiente de teste já citado em tópicos anteriores obtemos gráfico em barra dos tempos de execução(figura 7).

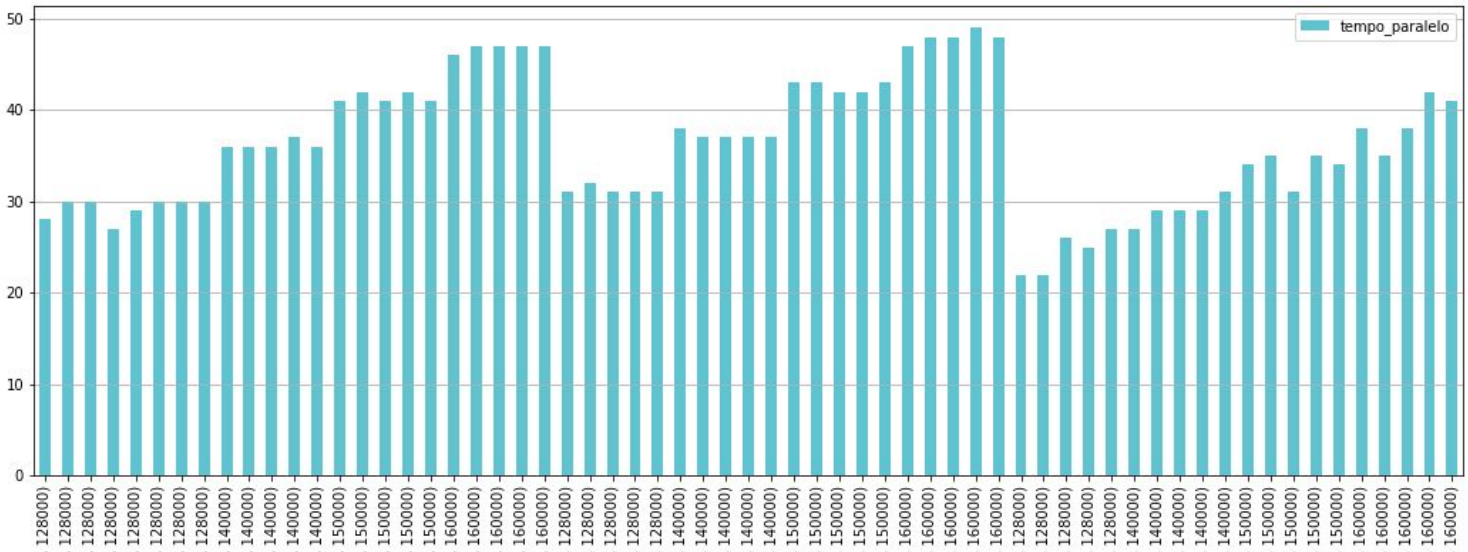
Figura 7- Tempos de execução código serial



Fonte -Autor

Para a execução do código paralelo temos geramos uma gráfico em barra para representar os tempos(Figura 8).

Figura 8- Tempos de execução código serial



Fonte -Autor

Realizamos a média das 5 execuções de cada situação em cada core e calculamos o Speedup (tempo serial/tempo paralelo) e a eficiência (Speedup/ números de processos) e geramos a tabela a seguir(figura 9).

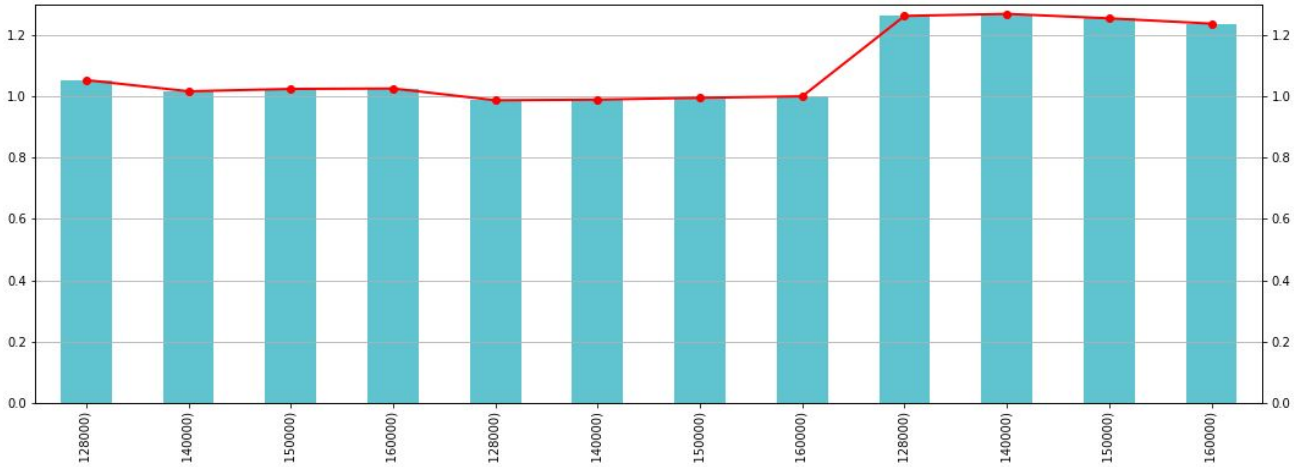
Figura 9- Tabela resultante do estudo

cores	problema	tempo_paralelo	tempo_serial	speedup	eficiencia
2	128000	29.25	30.8	1.052991	0.526496
2	140000	36.20	36.8	1.016575	0.508287
2	150000	41.40	42.4	1.024155	0.512077
2	160000	46.80	48.0	1.025641	0.512821
4	128000	31.20	30.8	0.987179	0.246795
4	140000	37.20	36.8	0.989247	0.247312
4	150000	42.60	42.4	0.995305	0.248826
4	160000	48.00	48.0	1.000000	0.250000
8	128000	24.40	30.8	1.262295	0.157787
8	140000	29.00	36.8	1.268966	0.158621
8	150000	33.80	42.4	1.254438	0.156805
8	160000	38.80	48.0	1.237113	0.154639

Fonte -Autor

Foi gerado um gráfico para demonstração do crescimento do speedup diante do aumento do cores no passar dos casos(figura 10).

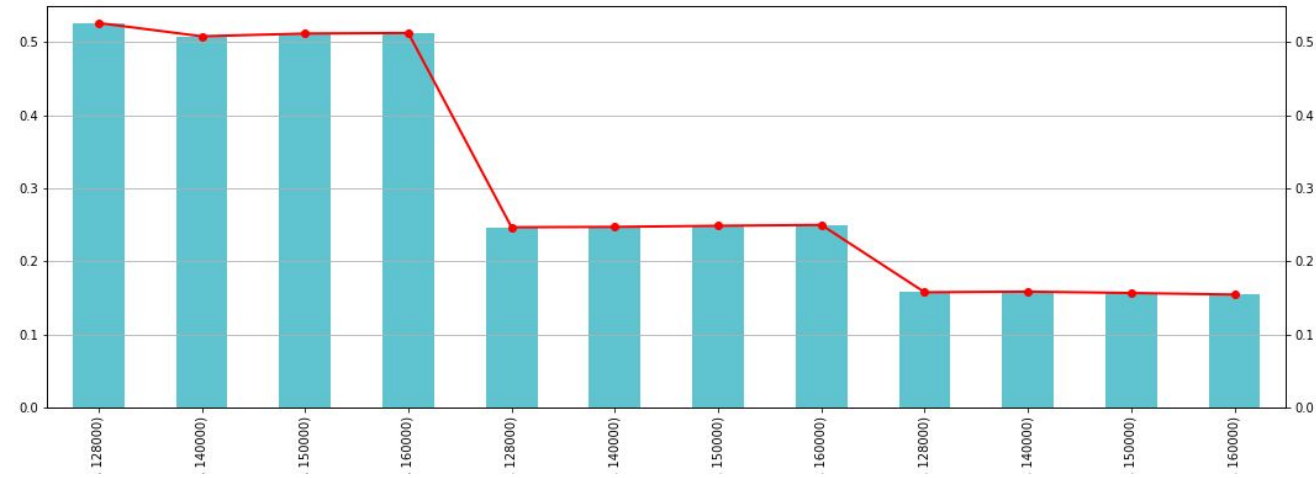
Figura 10- grafico do speed up



Fonte -Autor

Em uma visualização sobre o gráfico vemos um crescimento significativos somente a partir da utilização de 8 cores onde observamos um valores em torno de 1,26, que significa um ganho de 26% a menos no tempo de execução do código. Mas podemos observar que na utilização de 4 processos temos valores de execução igual ao em alguns casos piores que a execução serial.Também geramos uma gráfica para a eficiência(figura 11).

Figura 11- gráfico de eficiencia



Fonte -Autor

Para os casos de 2 e 4 cores vemos valores próximos aos 0,5 e 0,25, respectivamente, mostrando uma grande proximidade dos tempos ao tempo serial. Quando chegamos a utilização de 8 processos, mesmo estando longe da perfeição que seria uma velocidade 8 vezes maiores que o tempo normal, já visualizamos uma crescimento onde o valores iguais a seria estariam em 0,125 e obtivemos algo em torno de 0,158.

Considerações Finais

Em um resumo para uma estrutura do caso em paralelo não obtivemos ganhos reais com 2 e 4 processos e somente caso com 8 processos surgi um crescimento visível . Em um estudo mais aprofudado seria aconselhado um teste com a utilização mais cores/processos para visulizar se esse é o principal fator de crescimento e também um aumento no tamanho dos problemas para que possamos checar melhor o efeito da parte serial do nosso codigo paralelo.Para melhor visualizão os códigos (serial e paralelo) juntamente com o notebook em python, utilizado para realizar a analise dos dados e construção dos graficos, estão disponibilizados no github pelo link : https://github.com/PdrCarvalho/Odd_even_Transposition_Sort_parallelarrall .