

* 1. UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

MPI 2 - Odd-even Transposition Sort

**Pedro Cardoso Carvalho**

**NATAL-RN**

**2020**

**Sumário**

[**Introdução**](#_6xvfwkeumjbo) **3**

[**Desenvolvimento**](#_pkk2c8j6rvtm) **4**

[Ambiente de teste](#_jgh6kbg8vm3z) 4

[Programa serial](#_nclregthvvg7) 4

[Programa Paralelo](#_dh390irdhy5) 6

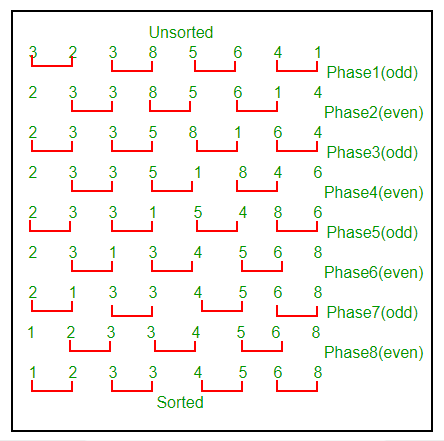
[Resultados](#_jybx269xcp5p) 8

[Considerações Finais](#_25qqtac1s9ie) **11**

# 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](https://www.geeksforgeeks.org/odd-even-sort-brick-sort/)

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

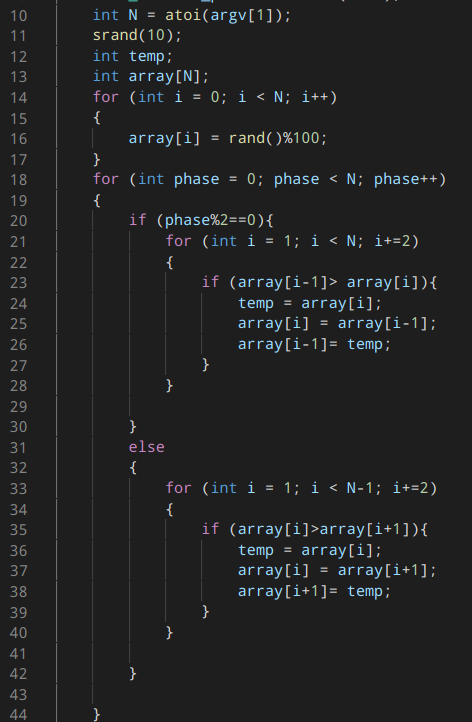
## Ambiente de teste

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).

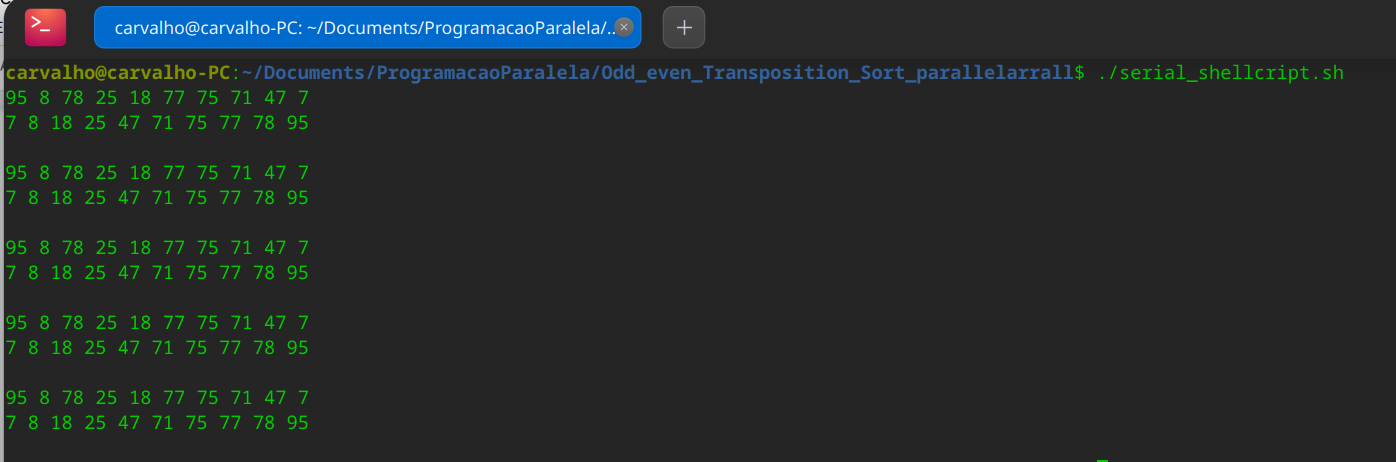
**Figura 2-** Codigo serial de Odd-even Transposition sort



**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

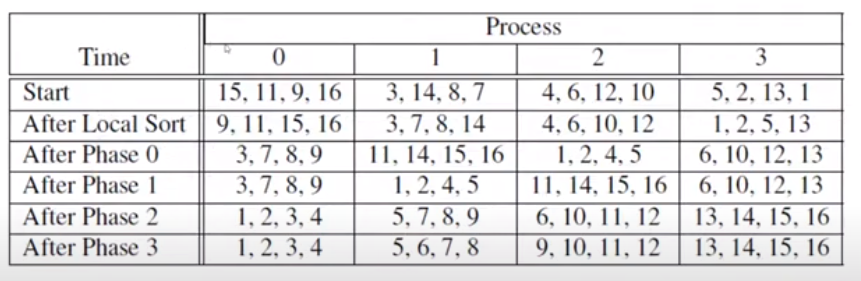


**Fonte -**Autor

## 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.

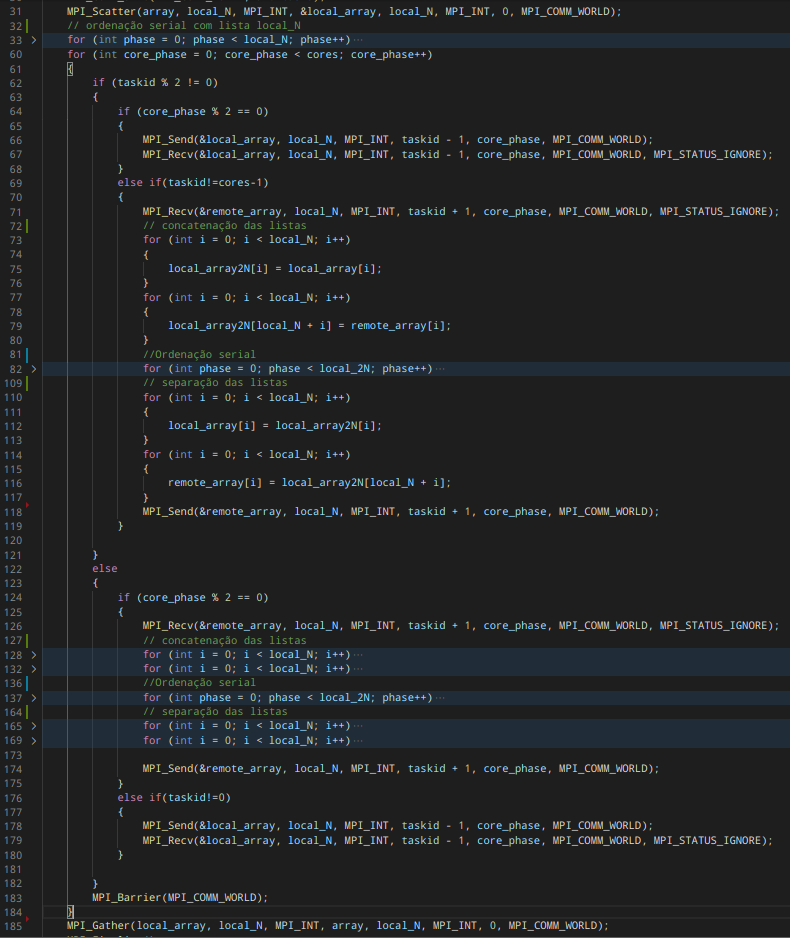
**Figura 4-** Estrategia do codigo paralelo para Odd-even Transposition sort



**Fonte -** Aula [Capítulo 3 - Aula 8 - MPI - Odd even Transposition Sort](https://www.youtube.com/watch?v=_qL4F4zAPF4&list=PLIs72eWiGzicSxa2jP75E8WxJVYvtFfmo&index=25&t=5s)

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.

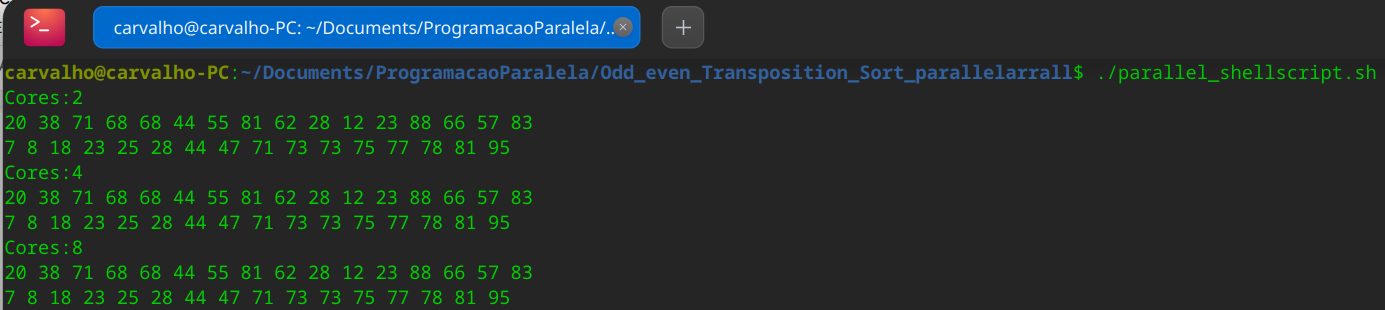
Para o código funcionar utilizamos as funções: MPI\_Scatter(para a divisão da lista igualmente para cada processo), MPI\_Gather( para união de todas as lista locais de processo por ordem de indices), MPI\_Barrier( para forçar uma sincronização dos processo ao final de cada momento par ou ímpar) e par 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

**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

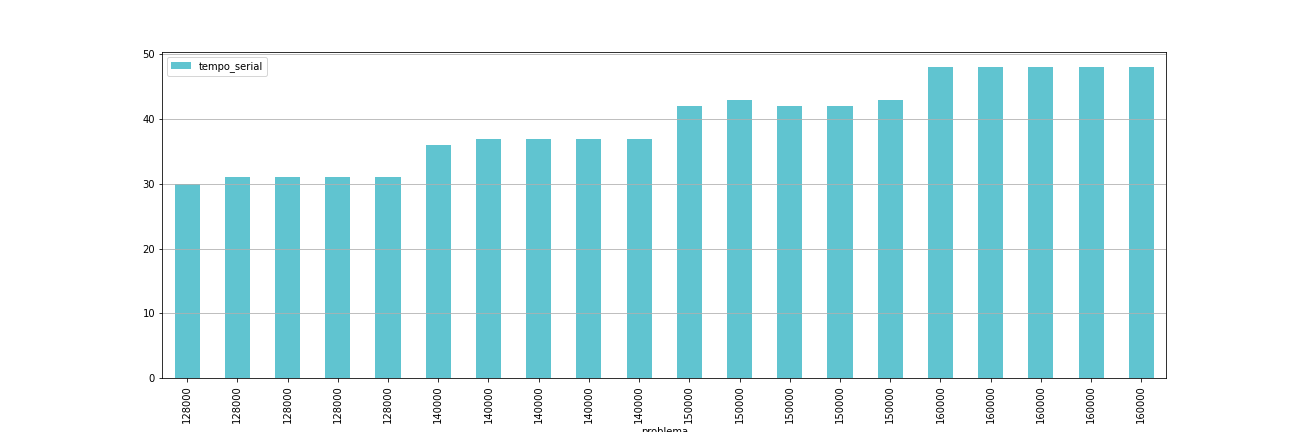


**Fonte -**Autor

## 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).

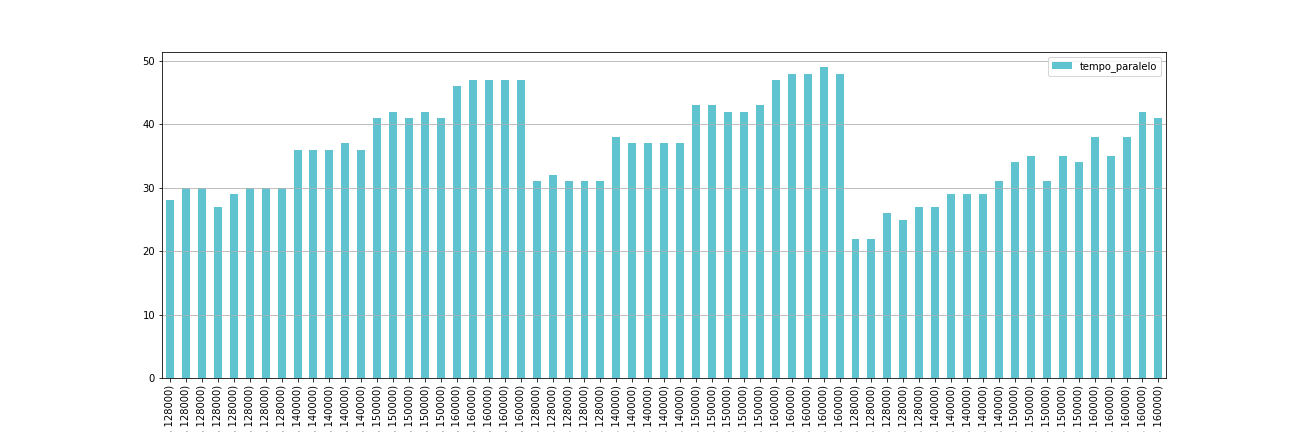
**Figura 7-** Tempos de execução código serial



**Fonte -**Autor

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

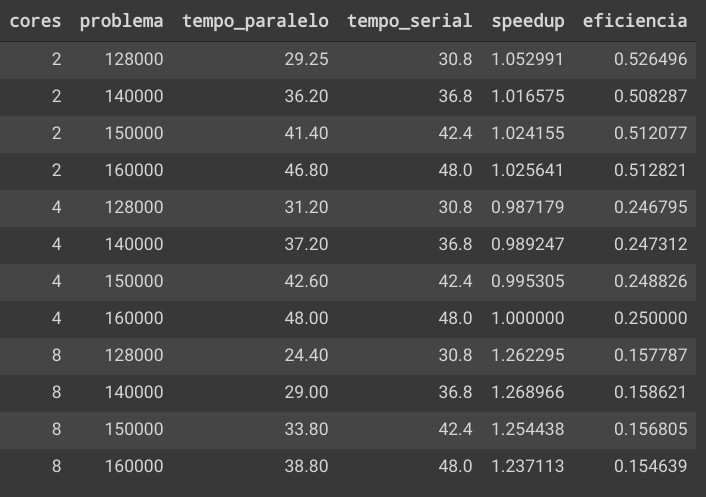
**Figura 8-** Tempos de execução código serial



**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).

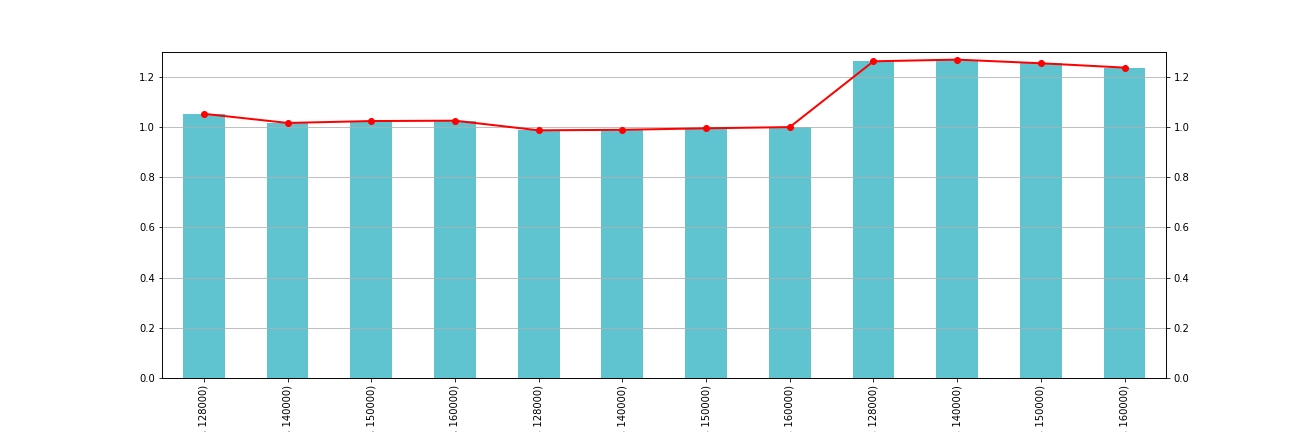
**Figura 9-** Tabela resultante do estudo



**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> .