Universidade Federal do Rio Grande do Norte

Instituto Metrópole Digital

Análise e Comparação de Speedup e Eficiência do código serial e paralelo no problema de ordenamento utilizando o algoritmo Odd-Even Transposition Sort

João Vitor Venceslau Coelho

Natal/RN

2020

Introdução

Este relatório consiste na explicação do problema do **ordenamento de um vetor de números**, buscando deixar claros todos os detalhes do problema e da solução utilizada, sendo esta: ***o algoritmo Odd-even Transposition Sort***. A versão serial e duas versões paralelas do algoritmo implementado serão explanadas, em seguida os resultados obtidos serão mostrados e comentados. Será também brevemente comentada a corretude dos algoritmos implementados e serão expostas as análises do speedup, da eficiência e da escalabilidade dos algoritmos paralelos para o serial. Ao final, um resumo das atividades e análises realizadas será exposto, sintetizando os principais pontos do relatório.

Ordenamento com Odd-even Transposition Sort

O **ordenamento de um vetor de números** é um problema bem conhecido e abordado na área de computação, consiste em reposicionar os elementos em um vetor de números de forma que fiquem em alguma ordem, comumente a ordem não-decrescente, dadas suas diversas aplicações e dependências para outras atividades, por exemplo, uma sequência ordenada facilita a busca de um elemento. Assim, algoritmos eficientes para ordenamento são bem desejados. O **Odd-even Transposition Sort** é um algoritmo de ordenação que pode ser facilmente paralelizado, ele se baseia, assim como o Bubble Sort, na ideia de trocar a posição de elementos vizinhos, a diferença é que ele realiza essas trocas seguindo dois momentos: compara a partir dos elementos pares, caso estejam na ordem errada, os troca; em outro momento esse procedimento é feito para os elementos ímpares, assim, ao alternar entre esses dois momentos a ordenação é realizada.

Desenvolvimento

Abaixo são explicadas as versões serial e paralelas do algoritmo **Odd-even Transposition Sort**. A parte do código relativa a contagem do tempo não está sendo indicada, mas apenas o tempo de execução da ordenação e da comunicação entre processos é considerada, o tempo que é gasto para alocar / desalocar a memória necessária e preencher o vetor com os números pseudo-aleatórios não foi contabilizado.

Soluções implementadas

Versão Serial do algoritmo Odd-even Transposition Sort

O algoritmo recebe como entrada três argumentos por linha de comando, são eles: a **seed**, usada para gerar os números pseudo-aleatórios; o **tamanho do vetor**, a ser ordenado; por último uma **flag** que indica se o resultado deve ser exibido ou não. Após inicializar a **seed**, é alocada a memória para o vetor de números a ser ordenado e em seguida ele é preenchido

com os números inteiros gerados com o rand. Com o vetor preenchido, dá-se início ao procedimento do Odd-even Transposition Sort, o primeiro laço contabiliza em qual momento o procedimento se encontra, momento par ou ímpar, para isso é utilizada a variável *phase* que ao obtermos o resto da divisão dela por 2, obtemos qual o momento correto, se o resto da divisão for 0 (momento par), inicia-se um laço a partir de 1 até n, com incremento 2, assim compara-se cada elemento de índice par com o seu elemento posterior, se esse elemento posterior for menor que o elemento de índice par, ocorre a troca. Caso o resto da divisão de *phase* por 2 seja 1 (momento ímpar), um laço semelhante ao do momento par é criado, porém ele vai até n-1, similarmente, a comparação entre o elemento de índice ímpar com o seu posterior é realizada, se o posterior for menor que o elemento de índice ímpar, a troca ocorre. Esses momentos ocorrem até o valor de *phase* ser igual ao tamanho do vetor e com isso o vetor está ordenado.

Versão Paralela do algoritmo Odd-even Transposition Sort

Os parâmetros necessários são os mesmo da versão serial: **seed**, **tamanho do vetor** e **flag** de exibição, após a leitura desses parâmetros o **MPI_Init** é chamado, o número de processos a serem usados é acessado e cada processo lê seu rank. Caso o número de processos solicitados não seja um divisor exato do tamanho do vetor informado, uma mensagem solicitando que o tamanho do vetor a ser ordenado seja dividido igualmente entre os processos é exibida pelo processo de rank 0, em seguida o **MPI_Finalize** é chamado finalizando a comunicação entre os processos e cada processo é finalizado. Caso o tamanho do vetor seja divisível pelo número de processos, o processo de rank 0 aloca o espaço em memória para o vetor a ser ordenado e o preenche com números gerados usando a **seed** informada. Em seguida, cada processo calcula qual o tamanho do vetor local (**local_n**) e aloca um vetor com o dobro do tamanho calculado, o processo 0 irá enviar para todos os processos a devida parte do vetor,. que foi preenchido com números pseudo-aleatórios usando o **MPI_Scatter**, cada processo irá preencher apenas os **local_n** primeiros números do vetor alocado anteriormente e ordená-los utilizando o mesmo procedimento do Odd-even Transposition Sort usado na versão serial. Após cada processo ter ordenado localmente seu conjunto de números, dá-se início às fases par-ímpar da versão paralela. Em cada fase os processos de rank par e ímpar irão enviar para seus vizinhos posteriores / anteriores o seu conjunto de números e receber o conjunto deles, para então ordenar o $2 \cdot$ **local_n** que possuem atualmente, após ordená-los, dependendo da fase atual do algoritmo. Se a fase for par, os processos pares recebem do seu vizinho posterior e mandam para o mesmo, assim, após realizar a ordenação, os números que cada processo deve manter são os **local_n** números iniciais do vetor ordenado, enquanto que os processos ímpares devem mover os **local_n** números no final do vetor ordenado para as **local_n** primeiras posições, caso essa fase seja ímpar a lógica se inverte, sendo os pares que devem mover os **local_n** números finais para as posições iniciais. Ao final de cada fase uma barreira é criada para forçar e manter os processos sincronizados nos seus envios e recebimentos, dessa forma,

quando todas as fases se encerram o **MPI_Gather** é chamado para unir, no espaço alocado pelo processo 0 (onde antes estavam os números desordenados), todos os devidos vetores de tamanho **local_n** que cada processo possui e, com isso, temos a ordenação do vetor. Por fim o processo de rank 0 irá mostrar na tela o tempo gasto para ordenar o vetor e, se a **flag** de exibição for 1, exibirá também o vetor ordenado, em seguida ele libera a memória alocada para guardar o vetor a ser ordenado e cada processo libera a memória do vetor usado para as ordenações locais que executaram. Com isso o **MPI_Finalize** é executado e os processos terminam.

Versão Paralela do algoritmo Odd-even Transposition Sort (Otimizada)

A versão paralela otimizada possui praticamente as mesmas etapas que a versão paralela descrita anteriormente, as diferenças se encontram nos seguintes pontos:

- O número de vetores alocados;
- O destino da chamada do **MPI_scatter**;
- O destino dos números recebidos do vizinho;
- A forma de ordenação nas fases par / ímpar;
- A origem usada na chamada do **MPI_gather**;
- O número de vetores desalocados;

A seguir são descritas cada uma das diferenças da versão otimizada em relação a outra: Sobre o número de vetores - Na versão otimizada cada processo aloca 3 vetores, dois de tamanho **local_n** (vetores **local_a** e **local_b**) e um de tamanho $2 \cdot$ **local_n** (vetor **aux**), em vez de apenas 1 vetor de tamanho $2 \cdot$ **local_n**, como na versão anterior; Em relação ao destino utilizado no **MPI_Scatter**, na versão otimizada é o vetor **local_a** que é utilizado como destino, sendo este o vetor de interesse do processo no lugar das **local_n** primeiras posições do vetor alocado na outra versão; Para o vetor recebido do vizinho, em cada fase par / ímpar é substituído o armazenamento nas **local_n** últimas posições, da versão não otimizada, pelo armazenamento no vetor **local_b**. No que diz respeito a forma de ordenamento utilizado nas fases par / ímpar, que é a principal distinção entre as versões, pois ao invés de aplicar novamente o Odd-Even Transposition Sort, como cada processo fez no vetor **local_a**, para realizar a ordenação nessas fases se executa um procedimento distinto descrito abaixo:

Iniciando a partir dos primeiros elementos de cada vetor, é comparado o elemento atual do primeiro vetor com o elemento atual do segundo vetor, após isso armazena-se o menor dos dois na menor posição livre do vetor **aux** e em seguida é alterado o elemento atual do vetor que possui esse menor elemento para o elemento seguinte; Dessa forma, o laço de repetição, presente no algoritmo, refaz esses passos enquanto, pelo menos, todos os elementos de um dos vetores não forem percorridos; Posteriormente, move-se os números restantes do vetor, que não teve todos os elementos utilizados, para as posições vagas em

aux e, com isso, o vetor **aux** possui a ordenação de todos os números dos dois vetores iniciais. Para finalizar esse procedimento de ordenação, os **local_n** primeiros elementos de **aux** são movidos para o primeiro vetor e os **local_n** últimos para o segundo vetor; Salienta-se que no procedimento descrito acima, o primeiro vetor é o **local_a** para os processos na fase atual (par ou ímpar) e **local_b** é o segundo vetor, enquanto que o contrário ocorre para o processo vizinho. Em relação às outras diferenças, na chamada do **MPI_Gather** todos os processos informam como origem o vetor **local_a**, em vez dos **local_n** primeiros elementos do vetor alocado e, sobre o número de vetores desalocados, os vetores **local_a**, **local_b** e **aux** são liberados, não sendo apenas o vetor de tamanho $2 \cdot$ **local_n** alocado pelo processo na versão anterior que é liberado;

Resultados encontrados

Os experimentos foram realizados numa máquina com a seguinte configuração:

- CPU: AMD® Ryzen 5 2600 six-core processor × 12
- Memória: 12 Gigabytes (11,7 GiB)
- Sistema Operacional (SO): Ubuntu 20.04.1 LTS - 64 bits

Corretude do Algoritmo Serial Odd-even Transposition Sort

Para ilustrar a corretude do algoritmo será apresentado um exemplo de execução:

Tempo	Estado do vetor
Início	15, 11, 09, 16, 03, 14, 08, 07, 04, 06, 12, 10, 05, 02, 13, 01
Fase 0	11, 15, 09, 16, 03, 14, 07, 08, 04, 06, 10, 12, 02, 05, 01, 13
Fase 1	11, 09, 15, 03, 16, 07, 14, 04, 08, 06, 10, 02, 12, 01, 05, 13
Fase 2	09, 11, 03, 15, 07, 16, 04, 14, 06, 08, 02, 10, 01, 12, 05, 13
Fase 3	09, 03, 11, 07, 15, 04, 16, 06, 14, 02, 08, 01, 10, 05, 12, 13
Fase 4	03, 09, 07, 11, 04, 15, 06, 16, 02, 14, 01, 08, 05, 10, 12, 13
Fase 5	03, 07, 09, 04, 11, 06, 15, 02, 16, 01, 14, 05, 08, 10, 12, 13
Fase 6	03, 07, 04, 09, 06, 11, 02, 15, 01, 16, 05, 14, 08, 10, 12, 13
Fase 7	03, 04, 07, 06, 09, 02, 11, 01, 15, 05, 16, 08, 14, 10, 12, 13
Fase 8	03, 04, 06, 07, 02, 09, 01, 11, 05, 15, 08, 16, 10, 14, 12, 13
Fase 9	03, 04, 06, 02, 07, 01, 09, 05, 11, 08, 15, 10, 16, 12, 14, 13
Fase 10	03, 04, 02, 06, 01, 07, 05, 09, 08, 11, 10, 15, 12, 16, 13, 14
Fase 11	03, 02, 04, 01, 06, 05, 07, 08, 09, 10, 11, 12, 15, 13, 16, 14
Fase 12	02, 03, 01, 04, 05, 06, 07, 08, 09, 10, 11, 12, 13, 15, 14, 16
Fase 13	02, 01, 03, 04, 05, 06, 07, 08, 09, 10, 11, 12, 13, 14, 15, 16
Fase 14	01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11, 12, 13, 14, 15, 16
Fase Final	01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11, 12, 13, 14, 15, 16

Tabela 1: Apresenta o estado do vetor a ser ordenado usando o algoritmo Odd-even Transposition Sort após a execução do mesmo em cada uma de suas fases. Em vermelho estão destacadas as trocas realizadas após a execução dessas fases.

Com esse exemplo é possível observar o resultado parcial obtido após cada fase do algoritmo serial, acompanhando, dessa forma, como a ordenação ocorre. Salienta-se ainda que, em cada uma dessa fases um laço de repetição existe, percorrendo os elementos do vetor para compará-los e trocá-los quando necessário. É interessante notar que nas fases ímpares o primeiro e o último elementos do vetor não participam das trocas.

Corretude do Algoritmo Paralelo Odd-even Transposition Sort

A mesma configuração inicial do vetor, utilizada no exemplo anterior, será usada para ilustrar as versões paralelas do algoritmo (não otimizada e otimizada), o resultado após a execução de cada fase é o mesmo para as duas versões, a diferença é apenas no modo de execução da ordenação, enquanto que na versão sem otimização em cada fase o procedimento Odd-Even Transposition Sort é executado, na versão otimizada ele ocorre apenas na ordenação local inicial de cada processo. Nas fases seguintes é usado o outro procedimento (descrito em Versão Paralela do algoritmo Odd-even Transposition Sort (Otimizada)) que utiliza a informação que os vetores de cada processo estão previamente ordenados, conseguindo assim diminuir o tempo gasto e a complexidade do algoritmo na versão paralela. Um pequeno exemplo comparando a execução das versões paralelas é mostrado logo depois, onde será possível perceber a diferença de como o ordenamento é realizado, assim como a diminuição de processamento obtido.

Tempo	Processos			
	0	1	2	3
Início	15, 11, 09, 16	03, 14, 08, 07	04, 06, 12, 10	05, 02, 13, 01
Depois do ordenamento local	09, 11, 15, 16	03, 07, 08, 14	04, 06, 10, 12	01, 02, 05, 13
Fase 0	03, 07, 08, 09	11, 14, 15, 16	01, 02, 04, 05	06, 10, 12, 13
Fase 1	03, 07, 08, 09	01, 02, 04, 05	11, 14, 15, 16	06, 10, 12, 13
Fase 2	01, 02, 03, 04	05, 07, 08, 09	06, 10, 11, 12	13, 14, 15 16
Fase 3	01, 02, 03, 04	05, 06, 07, 08	09, 10, 11, 12	13, 14, 15 16

Tabela 2: Apresenta o estado do vetor a ser ordenado em cada processo após a execução de cada umas das fases da versão paralela do algoritmo Odd-Even Transposition Sort.

Com o exemplo acima nota-se que os vetores do primeiro e do último processo, nem sequer mudam as posições de seus números, pois esses processos não participam das fases ímpares do algoritmo, estão aguardando a próxima fase (par) para realizarem algum ordenamento, enquanto que os processos entre eles praticamente sempre estão envolvidos em algum ordenamento.

Abaixo é exibida a execução da fase 0 em cada processo na versão paralela sem otimização, percebe-se que, na maioria das fases locais da fase 0, nenhuma troca sequer ocorre, gastando assim processamento apenas para percorrer o vetor e comparar seus valores, deixando o vetor igual ao início da fase, óbvio, pois isso é devido a organização inicial do vetor a ser ordenado e esse comportamento ocorre principalmente nos processos ímpares, devido a ordem que os números se encontram quando são unidos no vetor que é ordenado em cada processo.

Tempo	Processos			
	0	1	2	3
Após obter o vetor do vizinho	09, 11, 15, 16, 03, 07, 08, 14	03, 07, 08, 14, 09, 11, 15, 16	04, 06, 10, 12, 01, 02, 05, 13	01, 02, 05, 13, 04, 06, 10, 12
Fase 0: Fase 0 Local	09, 11, 15, 16, 03, 07, 08, 14	03, 07, 08, 14, 09, 11, 15, 16	04, 06, 10, 12, 01, 02, 05, 13	01, 02, 05, 13, 04, 06, 10, 12
Fase 0: Fase 1 Local	09, 11, 15, 03, 16, 07, 08, 14	03, 07, 08, 09, 14, 11, 15, 16	04, 06, 10, 01, 12, 02, 05, 13	01, 02, 05, 04, 13, 06, 10, 12
Fase 0: Fase 2 Local	09, 11, 03, 15, 07, 16, 08, 14	03, 07, 08, 09, 11, 14, 15, 16	04, 06, 01, 10, 02, 12, 05, 13	01, 02, 04, 05, 06, 13, 10, 12
Fase 0: Fase 3 Local	09, 03, 11, 07, 15, 08, 16, 14	03, 07, 08, 09, 11, 14, 15, 16	04, 01, 06, 02, 10, 05, 12, 13	01, 02, 04, 05, 06, 10, 13, 12
Fase 0: Fase 4 Local	03, 09, 07, 11, 08, 15, 14, 16	03, 07, 08, 09, 11, 14, 15, 16	01, 04, 02, 06, 05, 10, 12, 13	01, 02, 04, 05, 06, 10, 12, 13
Fase 0: Fase 5 Local	03, 07, 09, 08, 11, 14, 15, 16	03, 07, 08, 09, 11, 14, 15, 16	01, 02, 04, 05, 06, 10, 12, 13	01, 02, 04, 05, 06, 10, 12, 13
Fase 0: Fase 6 Local	03, 07, 08, 09, 11, 14, 15, 16	03, 07, 08, 14, 09, 11, 15, 16	01, 02, 04, 05, 06, 10, 12, 13	01, 02, 04, 05, 06, 10, 12, 13
Fase 0: Fase 7 Local	03, 07, 08, 09, 11, 14, 15, 16	03, 07, 08, 09, 11, 14, 15, 16	01, 02, 04, 05, 06, 10, 12, 13	01, 02, 04, 05, 06, 10, 12, 13
Subvetor do Processo	03, 07, 08, 09	11, 14, 15, 16	01, 02, 04, 05	06, 10, 12, 13

Tabela 3: Apresenta o estado do vetor a ser ordenado por processo durante as fases locais da fase 0.

É interessante notar que todas as trocas iniciam pelos elementos centrais já que os números antes e depois deles já foram ordenados por cada processo antes de enviarem / receberem os vetores para / de seus vizinhos. Talvez uma possível pequena melhoria para essa versão seria remover esse ordenamento local que cada processo realiza no seu vetor, pois em seguida outro ordenamento é feito utilizando esses mesmo números, ordenando-os novamente.

A execução da fase 0 na versão otimizada é ilustrada a seguir, os números destacados em azul são os valores que estão sendo comparados atualmente, o vetor auxiliar é onde o resultado do ordenamento é armazenado e os valores em vermelho, em cada vetor, representam os números que já foram ordenados. Vale destacar que, na tabela de exemplo da execução da versão sem otimização, era exposto apenas o resultado após uma execução

do laço de repetição mais externo do algoritmo, isto é, após cada fase par / ímpar, e em cada uma dessas fases, um laço interno é executado, percorrendo o vetor e comparando números vizinhos. Enquanto que na tabela abaixo os resultados não dependem de outro laço interno, dependem apenas da comparação que é feita naquele momento, diminuindo, assim, o número de operações realizadas, mesmo que a tabela aqui apresentada seja maior.

Tempo	Processos							
	0		1		2		3	
Vetores A e B Iniciais	09, 11, 15, 16	03, 07, 08, 14	03, 07, 08, 14	09, 11, 15, 16	04, 06, 10, 12	01, 02, 05, 13	01, 02, 05, 13	04, 06, 10, 12
Vetor Auxiliar Inicial								
1 - Vetores A e B	09, 11, 15, 16	03, 07, 08, 14	03, 07, 08, 14	09, 11, 15, 16	04, 06, 10, 12	01, 02, 05, 13	01, 02, 05, 13	04, 06, 10, 12
1 - Vetor Auxiliar	03		03		01		01	
2 - Vetores A e B	09, 11, 15, 16	03, 07, 08, 14	03, 07, 08, 14	09, 11, 15, 16	04, 06, 10, 12	01, 02, 05, 13	01, 02, 05, 13	04, 06, 10, 12
2 - Vetor Auxiliar	03, 07		03, 07		01, 02		01, 02	
3 - Vetores A e B	09, 11, 15, 16	03, 07, 08, 14	03, 07, 08, 14	09, 11, 15, 16	04, 06, 10, 12	01, 02, 05, 13	01, 02, 05, 13	04, 06, 10, 12
3 - Vetor Auxiliar	03, 07, 08		03, 07, 08		01, 02, 04		01, 02, 04	
4 - Vetores A e B	09, 11, 15, 16	03, 07, 08, 14	03, 07, 08, 14	09, 11, 15, 16	04, 06, 10, 12	01, 02, 05, 13	01, 02, 05, 13	04, 06, 10, 12
4 - Vetor Auxiliar	03, 07, 08, 09		03, 07, 08, 09		01, 02, 04, 05		01, 02, 04, 05	
5 - Vetores A e B	09, 11, 15, 16	03, 07, 08, 14	03, 07, 08, 14	09, 11, 15, 16	04, 06, 10, 12	01, 02, 05, 13	01, 02, 05, 13	04, 06, 10, 12
5 - Vetor Auxiliar	03, 07, 08, 09, 11		03, 07, 08, 09, 11		01, 02, 04, 05, 06		01, 02, 04, 05, 06	
6 - Vetores A e B	09, 11, 15, 16	03, 07, 08, 14	03, 07, 08, 14	09, 11, 15, 16	04, 06, 10, 12	01, 02, 05, 13	01, 02, 05, 13	04, 06, 10, 12
6 - Vetor Auxiliar	03, 07, 08, 09, 11, 14		03, 07, 08, 09, 11, 14		01, 02, 04, 05, 06, 10		01, 02, 04, 05, 06, 10	
7 - Vetores A e B	09, 11, 15, 16	03, 07, 08, 14	03, 07, 08, 14	09, 11, 15, 16	04, 06, 10, 12	01, 02, 05, 13	01, 02, 05, 13	04, 06, 10, 12
7 - Vetor Auxiliar	03, 07, 08, 09, 11, 14, 15, 16		03, 07, 08, 09, 11, 14, 15, 16		01, 02, 04, 05, 06, 10, 12		01, 02, 04, 05, 06, 10, 12	
8 - Vetores A e B					04, 06, 10, 12	01, 02, 05, 13	01, 02, 05, 13	04, 06, 10, 12
8 - Vetor Auxiliar					01, 02, 04, 05, 06, 10, 12, 13		01, 02, 04, 05, 06, 10, 12, 13	
Subvetor do Processo	03, 07, 08, 09		11, 14, 15, 16		01, 02, 04, 05		06, 10, 12, 13	

Tabela 4: Apresenta os estados dos vetores que estão sendo ordenados e do vetor auxiliar, por processo, durante a fase 0. Os vetores A, B e o auxiliar no passo 8 do ordenamento nos processos 0 e 1 não foram preenchidos pois esse passo não ocorreu nesses processos, pois os valores 15 e 16 foram apenas copiados para o vetor auxiliar depois do passo 7.

A execução das fases seguintes ocorre de forma similar, não é mostrado aqui a execução completa, pois não é esse o objetivo e sim apenas ilustrar parte da execução do algoritmo.

Com esses exemplos da execução das versões paralelas do algoritmo Odd-Even Transposition Sort, espera-se ter sido mostrada a corretude dos mesmos.

Análise de Speedup, Eficiência e Escalabilidade

Abaixo estão as tabelas com as médias dos tempos válidos obtidos, sendo que para cada instância analisada, foram coletados 15 tempos, em que os dois maiores e os dois menores foram desconsiderados como válidos para o cálculo dessa média. A seed utilizada para gerar os números pseudo-aleatórios foi 0 e o tamanho do problema consiste no tamanho do vetor ser ordenado. Abaixo estão as médias dos tempos que cada ordenação demorou:

Problema	Serial	2	3	4	6	8	12
126000	32,1283590	29,4042821	33,4341503	24,9404014	17,8059623	19,8987508	14,1579596
131400	35,2632687	31,9982775	36,3748106	27,1318706	18,8331079	21,2556747	15,4238674
136800	38,2338163	34,7281548	39,4498532	29,4153286	20,4050331	23,6022377	16,6140388
142200	41,1952384	37,5043416	42,6298033	31,7566855	22,0838846	24,9690377	18,0200765
147600	44,4954956	40,4136832	45,9411623	34,2454442	23,7887748	27,1845029	19,3588099
153000	47,7939913	43,4486088	49,3718146	36,7675086	25,5235092	29,1789551	20,7334404
158400	51,2313681	46,6121026	52,9349470	39,4264354	27,3773716	31,2884170	22,2903886
163800	54,8739602	49,8463059	56,6216804	42,1748635	29,3360994	32,3956991	23,9238401
169200	59,0652826	53,2480959	60,4046619	45,0005467	31,3957319	35,2891812	25,5451760
174600	63,4405412	56,7206768	64,3557493	47,9312321	33,5071291	36,9082017	27,0387866
180000	68,0287583	60,2640256	68,4133966	50,9564279	35,5266549	39,6182850	28,9526963

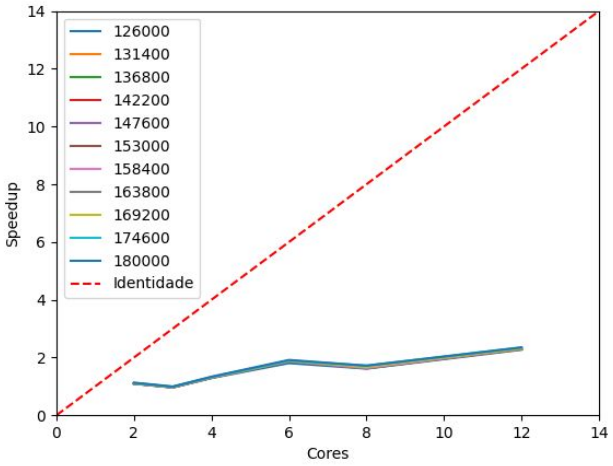
Tabela 5: Apresenta os tempos médios obtidos nos experimentos por tamanho do problema e algoritmo utilizado, assim como a quantidade de cores na versão paralela sem otimização.

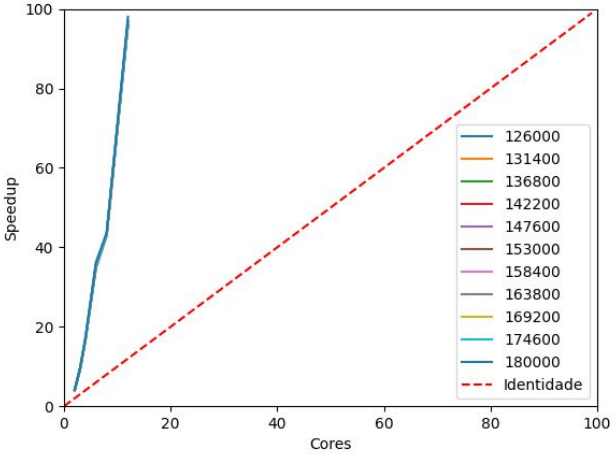
Problema	Serial	2	3	4	6	8	12
126000	32,1283590	7,9457249	3,4670878	1,9323012	0,9241188	0,7467723	0,3355711
131400	35,2632687	8,6437547	3,7737490	2,1060757	1,0112712	0,8157897	0,3645240
136800	38,2338163	9,3813503	4,0982093	2,2890332	1,0758274	0,8821958	0,3968041
142200	41,1952384	10,1655297	4,4413172	2,4794104	1,1429112	0,9543753	0,4266669
147600	44,4954956	10,9641691	4,7819034	2,6700412	1,2670948	1,0347782	0,4598581
153000	47,7939913	11,7969687	5,1593516	2,8777330	1,3526494	1,1061848	0,4951396
158400	51,2313681	12,6491081	5,5431701	3,0788481	1,4623450	1,1875992	0,5308426
163800	54,8739602	13,5740134	5,9357321	3,3135328	1,5544917	1,2767589	0,5709200
169200	59,0652826	14,5334633	6,3702996	3,5260555	1,6465457	1,3663909	0,6107452
174600	63,4405412	15,4774330	6,7732863	3,7696395	1,7889603	1,4661970	0,6531037
180000	68,0287583	16,4650239	7,1969920	4,0183839	1,8774350	1,5574387	0,6939633

Tabela 6: Apresenta os tempos médios obtidos nos experimentos por tamanho do problema e algoritmo utilizado, assim como a quantidade de cores na versão paralela com otimização.

A diferença entre os tempos obtidos é extremamente discrepante, a versão otimizada diminui muito o tempo médio gasto na execução do algoritmo.

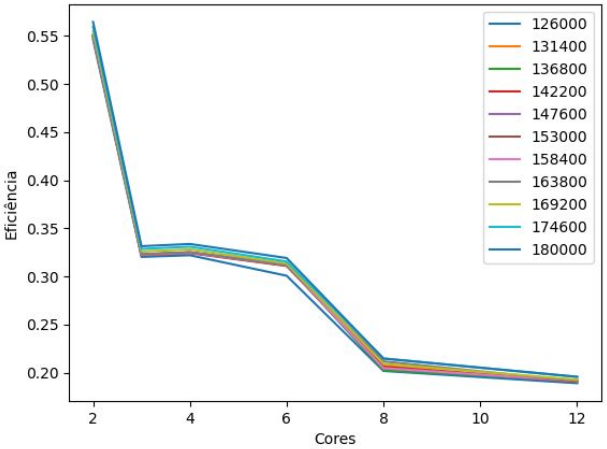
Abaixo seguem as análises e comparações de speedup e eficiência das versões paralelas do algoritmo de ordenação Odd-Even Transposition Sort e seus devidos gráficos.

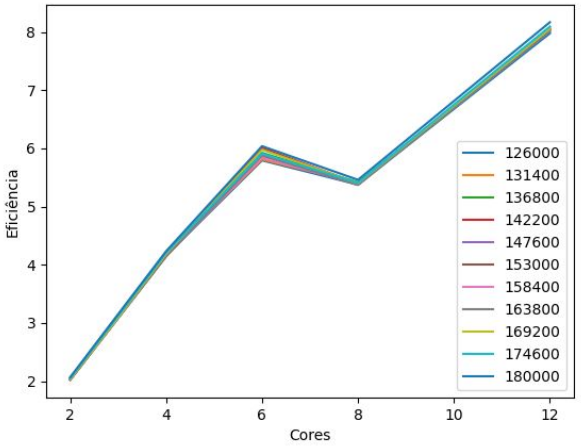




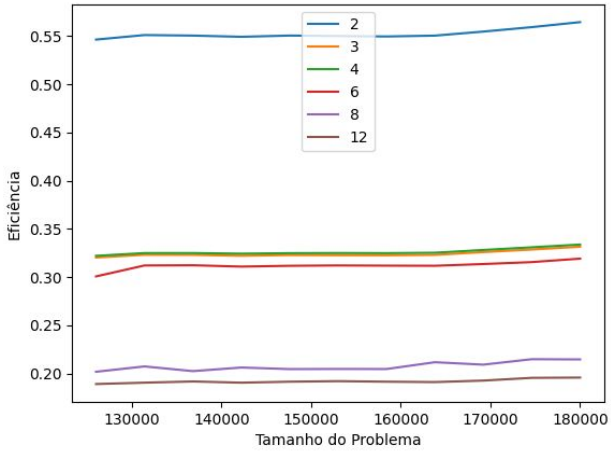
Imagens 1 e 2: Comparações do speedup em cada um dos tamanhos do problema com o speedup ideal. Na imagem 1 o speedup incrementa muito pouco com o aumento dos cores e é perceptível uma diminuição do speedup ao serem usados 3 e 8 cores. Enquanto na imagem 2 o speedup cresce muito rapidamente, porém ainda percebe-se uma "quebra" no ritmo do crescimento ao utilizar 8 cores .

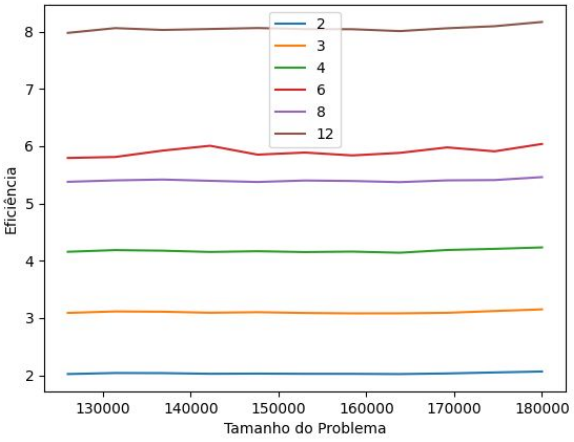
Acredita-se que a primeira diminuição na imagem 1 seja devido ao número ímpar de processos utilizados, o que impacta na performance do algoritmo paralelo, enquanto a diminuição com o uso de 8 cores nas duas imagens seja devido ao uso dos cores lógicos da máquina. Que possuem um desempenho inferior aos cores físicos.





Imagens 3 e 4: Apresentam a eficiência por cores em cada tamanho de problema utilizado. As linhas obtidas possuem comportamentos bem diferentes, com o algoritmo não otimizado a tendência é a diminuição da eficiência com o aumento dos cores e ocorre uma queda abrupta com o uso dos cores lógicos, chegando a 0.2 de eficiência com 12 cores. Enquanto com o algoritmo otimizado a tendência é um crescimento da eficiência com o aumento de cores, mas, ainda assim, é perceptível a queda de performance com o uso de cores lógicos, porém ainda alcança uma eficiência de quase 8 no final.





Imagens 5 e 6: Na comparação da eficiência por tamanho do problema, de acordo com a quantidade de cores utilizados, percebe-se novamente o comportamento inverso entre os gráficos, mesmo que com um mesmo número de cores e com o aumento do tamanho do problema não exista uma variação na eficiência, a mesma permaneça praticamente constante, é visível a diferença do valor da eficiência com diferentes números de cores.

Diante desses dados, é concluído que a versão sem otimização do algoritmo paralelo se categoriza como **Fracamente Escalável**, pois aumentando o tamanho do problema e o número de cores em proporções iguais, a eficiência se mantém constante, com algumas pequenas flutuações. Enquanto que a versão otimizada, dados os gráficos exibidos, é **Fortemente Escalável,** pois para um mesmo tamanho de problema, com o aumento do número de cores utilizados ocorre o aumento da eficiência.

Considerações Finais

Neste relatório apresentou-se brevemente o problema da ordenação de um vetor de números e o que é o algoritmo de ordenação Odd-Even Transposition Sort . Foram explanadas as versões serial e duas versões paralelas do mesmo algoritmo, sendo uma delas otimizada para usufruir da ordenação parcial que é realizada durante o processo de paralelização das tarefas, obtendo assim uma performance superior na questão de tempo gasto durante a execução.

Após declarar qual a configuração do hardware da máquina utilizada nos experimentos, foi realizada uma breve apresentação da corretude dos algoritmos implementados, exemplificando a execução dos mesmos por meio de tabelas com os resultados obtidos a cada passo realizado pelos algoritmos. Na versão serial foram destacadas as trocas realizadas em cada uma das fases do algoritmo para a instância utilizada do problema. Para as versões paralelas foi exemplificado o comportamento geral das duas versões, que é o mesmo, sendo a diferença entre elas apenas a execução interna das fases pares e ímpares do algoritmo e essa execução interna das fases foi exemplificada mostrando o passo a passo realizado por ambas as versões durante a fase 0 da mesma instância do problema.

A versão não otimizada acaba realizando muitas checagens no vetor a ser ordenado, muitas vezes não executando troca alguma de elementos, apenas confirmando que o vetor que está sendo ordenado, de fato está ordenado, enquanto a versão otimizada utiliza da ordenação parcial realizada previamente por cada processo para agilizar o ordenamento total do vetor alvo, diminuindo consideravelmente o processamento realizado e consequentemente o tempo gasto na tarefa.
Em seguida é feita a análise e comparação do speedup e eficiências dos algoritmos paralelos implementados, expondo os tempos médios obtidos nos experimentos realizados e discutindo os gráficos de speedup e eficiência gerados por esses tempos. É facilmente perceptível o ganho em performance da versão otimizada em relação a versão não otimizada apresentada, assim como o impacto que o uso dos cores lógicos da máquina têm sob a execução dos algoritmos, principalmente nos gráficos de eficiência. Por fim é feita a categorização das versões paralelas em relação a escalabilidade delas, sendo a versão não otimizada Fracamente Escalável e a otimizada Fortemente Escalável.

Referências

Introdução a Sistemas Paralelos

Odd–even sort

Odd Even Transposition Sort / Brick Sort using pthreads

Odd Even Transposition Sort - MPI Programming

Algoritmos Paralelos - ordenação

Paralelizar algoritmo de ordenação Odd-even sort em python