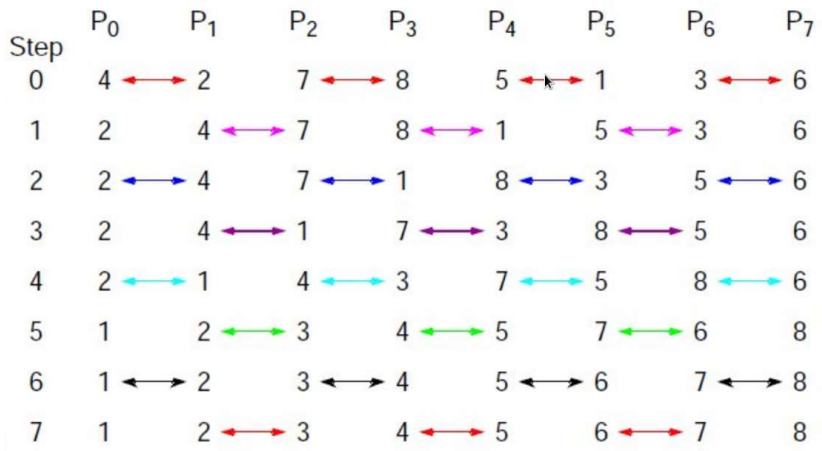
Odd-Even Transposition Sort - Implementação Serial e Paralela em linguagem C utilizando MPI

Leandro Silva Ferreira Junior

1. Introdução

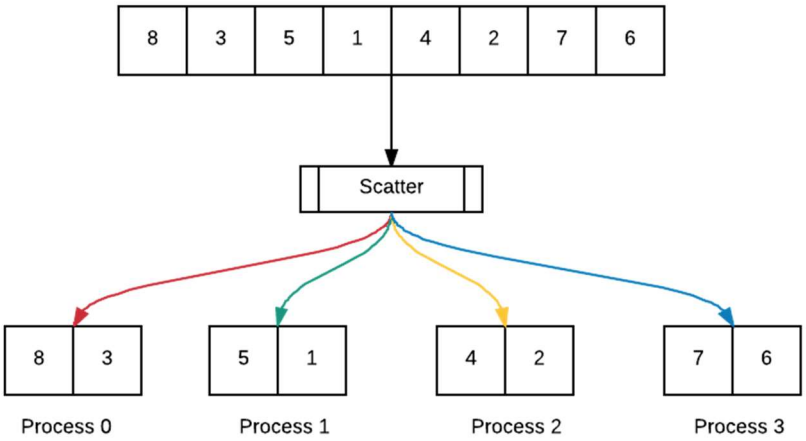
O Odd-Even Sort Transposition é um método de ordenação de vetores derivado do método Bubble Sort, e consiste na comparação de duplas de números em duas fases diferentes. Quando aplicado de maneira serial, durante a fase odd, os elementos ímpares do vetor são comparados com os elementos seguintes, e durante a fase even os elementos pares são comparados com seus respectivos elementos seguintes. Durante as comparações, se o primeiro da dupla for maior, os números são trocados. Após n fases de troca, onde n é o tamanho do vetor, a sequência estará ordenada:



(imagem 1)

Para tamanhos de problemas, ou no caso, vetores muito grandes, a quantidade de comparações aumenta de maneira exponencial, o que torna o tempo de execução dos programas muito alto. Buscando resolver o fator tempo de execução do ordenamento, uma das alternativas é a paralelização do algoritmo, que neste caso será feita usando a arquitetura de memória local dos núcleos de processamento.

Essa paralelização e controle da arquitetura é feita utilizando a biblioteca MPI em linguagem C, onde dessa forma podemos dividir o vetor para n núcleos de processamento diferentes, assim é possível comparar n duplas de números ao mesmo tempo em cada uma das fases Even-Odd.



(imagem 2)

Neste algoritmo, dentro de cada pthread, inicialmente seu vetor local será ordenado utilizando a função odd-even. Após o ordenamento interno, a função odd-even será aplicada aos processos, desta forma, cada pthread irá interagir alternando com a sua próxima e com a pthread anterior, comparando e ordenando os números entre si.

Ao final, basta que os vetores locais sejam unidos na ordem correta para que tenhamos como resultado o vetor original inicialmente definido porém perfeitamente ordenado.

O resultado esperado é que, quanto maior for o número de núcleos de processamento utilizado, mais rápida seja a execução do programa. A performance diminui quando o número de partes do vetor geradas é maior que o de processadores por gerar atividade multitask na CPU. Ao final discutiremos os resultados práticos de speedup e eficiência.

2. Desenvolvimento

2.1 Implementações

2.1.1 Implementação Serial (linguagem C)

Para a implementação em serial, foram definidas 3 funções no algoritmo: A função Fill, OddevenSortLocal, e Print.

A função Fill tem o objetivo de preencher o vetor a ser ordenado, e utiliza a Rand da linguagem C para gerar valores inteiros entre 0 e 10.000. O vetor e seu tamanho são passados por parâmetro e foi incluída uma seed fixa para padronizar os vetores gerados a cada execução do programa.

A função OddevenSortLocal abriga o ordenamento do vetor. Inicialmente são declaradas duas variáveis: um inteiro auxiliar para as trocas dos números, e um booleando com valor “false” que será utilizado como controle do ordenamento. Os ordenamentos de fase odd

e even estão inseridos neste laço cujo objetivo é ser executado enquanto o vetor não estiver totalmente ordenado, e para isso foi utilizada a estrutura “while” e o booleano de controle. Como na primeira execução o booleano é falso, o laço se inicia e já atribui o valor true, para que se nenhuma troca seja feita, o laço não seja executado novamente.

A estrutura for é utilizada nas fases odd-even onde a única diferença entre a fase odd e a fase even é o contador inicial i que indica a posição inicial do vetor, 1 para a fase Odd e 0 para a fase Even, o que garante que cada uma verifique apenas os valores de posição ímpares e pares, respectivamente, além disso, os laços vão até n-2 para que o último elemento não tente comparar com um elemento seguinte que não existe, e o contador é i+2 para executar a comparação à cada dupla.

Dentro do laço é verificado se o número em análise é maior que o seguinte, e se for o caso, eles são trocados com a ajuda da variável auxiliar. Se a troca for realizada, o booleano recebe novamente o valor de falso para que o laço entenda que o vetor ainda não está completamente ordenado e execute o ordenamento mais uma vez.

A função Print foi implementada recebendo o vetor e seu tamanho como parâmetros. Nela, inicialmente imprimimos uma quebra de linhas, e após, um laço percorre todo o vetor imprimindo os seus valores. Quando o código é executado no supercomputador, utilizando o shellscript, esta função imprime os vetores no arquivo “serial_oddeven.out”, o qual foi utilizado para averiguar se o algoritmo está correto.

A função main, encarregada da execução principal, inicialmente recebe um parâmetro que é atribuído como um valor inteiro à variável n que indica o tamanho do vetor, em seguida é declarado um vetor com o tamanho n e utilizando malloc para o alocamento dinâmico deste vetor, pois só saberemos seu tamanho pelo parâmetro passado. Após esse alocamento, declaramos a estrutura timeval para calcular o tempo de ordenamento.

As funções são executadas de forma que o tempo que calculamos se resume apenas à ordenação dos elementos. Após essa etapa, a main abre o arquivo txt e imprime nele o tempo de execução do ordenamento, e por último, desaloca o vetor dinâmico usando a função free.

2.1.2 Implementação Paralela (Utilizando MPI):

Para realizar a paralelização, dentro da main teremos algumas alterações. É definida além do que já consta na main serial, uma variável de inteiro para guardar o rank de cada pthread, essa variável é de suma importância para a comunicação dos processos.

A primeira etapa consiste em abrir a região paralela usando MPI_Init e alocar os ranks de cada pthread, em seguida criamos uma condição para que apenas o rank 0 preencha o vetor principal e grave o tempo inicial, evitando conflito entre as threads. As threads, por sua vez chamam a função oddevenSortGlobal, responsável por todo o ordenamento, e esta já retorna o vetor final ordenado.

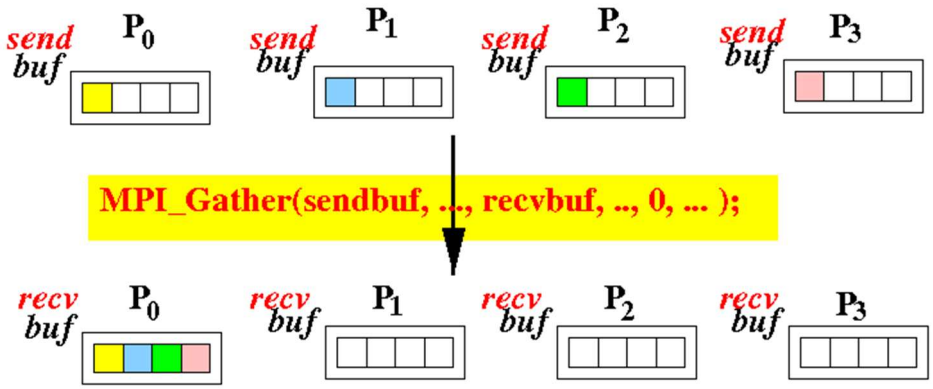
Após esta etapa novamente é definida uma condição para que somente o rank 0 grave o tempo final, imprima o vetor no .out, e grave no arquivo paralelo_oddeven_time.txt o tempo de execução. Por último, a main executa o MPI_Finalize para fechar a região paralela e desaloca o vetor dinâmico principal.

A função oddevenSortGlobal, é encarregada de todo o ordenamento. Para isso, declaramos as variáveis rank da pthread, o número total de phtreads e um contador para as fases; além disso foi alocado dinamicamente um vetor local para cada pthread, onde o seu tamanho é exatamente o tamanho do vetor principal dividido pelo número de processos.

A partir deste ponto, chamamos a função MPI_Scatter, a qual envia para os processos as suas respectivas partes do vetor e aloca-as no vetor local de cada uma (imagem 2), então, é possível chamarmos a função oddevenSortLocal para ordenar o vetor local de cada pthread.

O laço seguinte define as fases de comunicação entre os processos, e dentro são utilizadas duas condições: a primeira garante que o rank de mesma natureza da fase, odd ou even, execute o seu conteúdo, que é chamar a função sortPthreads trocando dados com o rank seguinte; já a segunda condição garante que o rank com natureza diferente da fase execute o seu conteúdo, que é chamar a função sortPthreads trocando dados com o rank anterior.

Finalmente, a oddevenSortGlobal executa o MPI_Gather, o qual devolve ao vetor global os seus dados ordenados como mostrado na imagem a seguir:



(imagem 3)

A função sortPthreads recebe como parâmetros principais o sendrank e o recvrank. Nesta função, são definidos um vetor auxiliar(aux) de mesmo tamanho do vetor local, e um segundo vetor (mergedVetor) com o dobro do tamanho do vetor local, são definidas também duas constantes que serão utilizadas como tags de comunicação entre cada par de MPI_Send e MPI_Recv: sendVetor e givebackVetor.

Se o rank que está executando a função for igual ao sendrank, então ele é o rank à esquerda do par, e executará um MPI_Send que envia ao rank à direita o vetor local, e um MPI_Recv que recebe do rank à direita o seu vetor local de resposta.

Caso contrário, então o rank em questão é o que está a direita. Este é responsável por receber do rank à esquerda o seu vetor local utilizando MPI_Recv, atribuí-lo em seu vetor auxiliar, e então uní-los utilizando a função merge que atribui o vetor fundido no mergedVetor. Em seguida é chamada a função oddevenSortLocal para ordenar o mergedVetor. São atribuídas então duas constantes que endereçam o início e o meio do vetor fundido. A constante do início (theirstart) é então utilizada como endereço no MPI_Send, que envia de volta ao rank da esquerda a primeira parte do mergedVetor. Por último um laço que começa na constante com

o endereço do meio do vetor fundido(mystart) atribui a segunda parte do mergedVetor ao próprio vetor local do rank.

A função merge, como citada anteriormente, une dois vetores em um terceiro. Para isso ela recebe os dois vetores e seus tamanhos, além do vetor de saída como parâmetros. Nela um contador é definido para atribuir o primeiro vetor de entrada à primeira parte do vetor de saída, e outro laço atribui o segundo vetor de entrada à segunda parte do vetor de saída.

2.2. Resultados Experimentais

2.2.1 Configuração do problema

O testes foram realizados para 4 tamanhos de vetores diferentes: 13 mil, 32 mil , 64 mil e 128 mil pontos. Para cada tamanho, o código foi executado 15 vezes, e o tempo de execução que utilizaremos consiste da média desses 15 tempos.

Para gerar a execução do código de forma organizada, foi utilizado um shellscript no supercomputador que carrega todos os parâmetros necessários para essa execução em um nó exclusivo da máquina, a compilação do código, e o laço para os tamanhos diferentes e as 15 execuções. Além disso o shellscript imprime no arquivo txt uma tabela com o número de cores utilizados, e o tamanho utilizado a cada 15 execuções, das quais o próprio programa escreve no mesmo arquivo o tempo de cada uma.

No caso do script paralelo, foi inserido mais um laço para o número de cores, desta forma, os testes para os 4 tamanhos serão realizados utilizando 2, 4, 8, 16 e 32 cores.

Dessa forma, o código C é executado já considerando o número de cores utilizados, e utiliza esse parâmetro quando inicia a região paralela, assim, abre o número de processos igual ao número de cores utilizados. No caso de ser executado com 2 cores, o programa abre 2 processos, e consequentemente divide o vetor a ser analisados em duas partes, e assim sucessivamente.

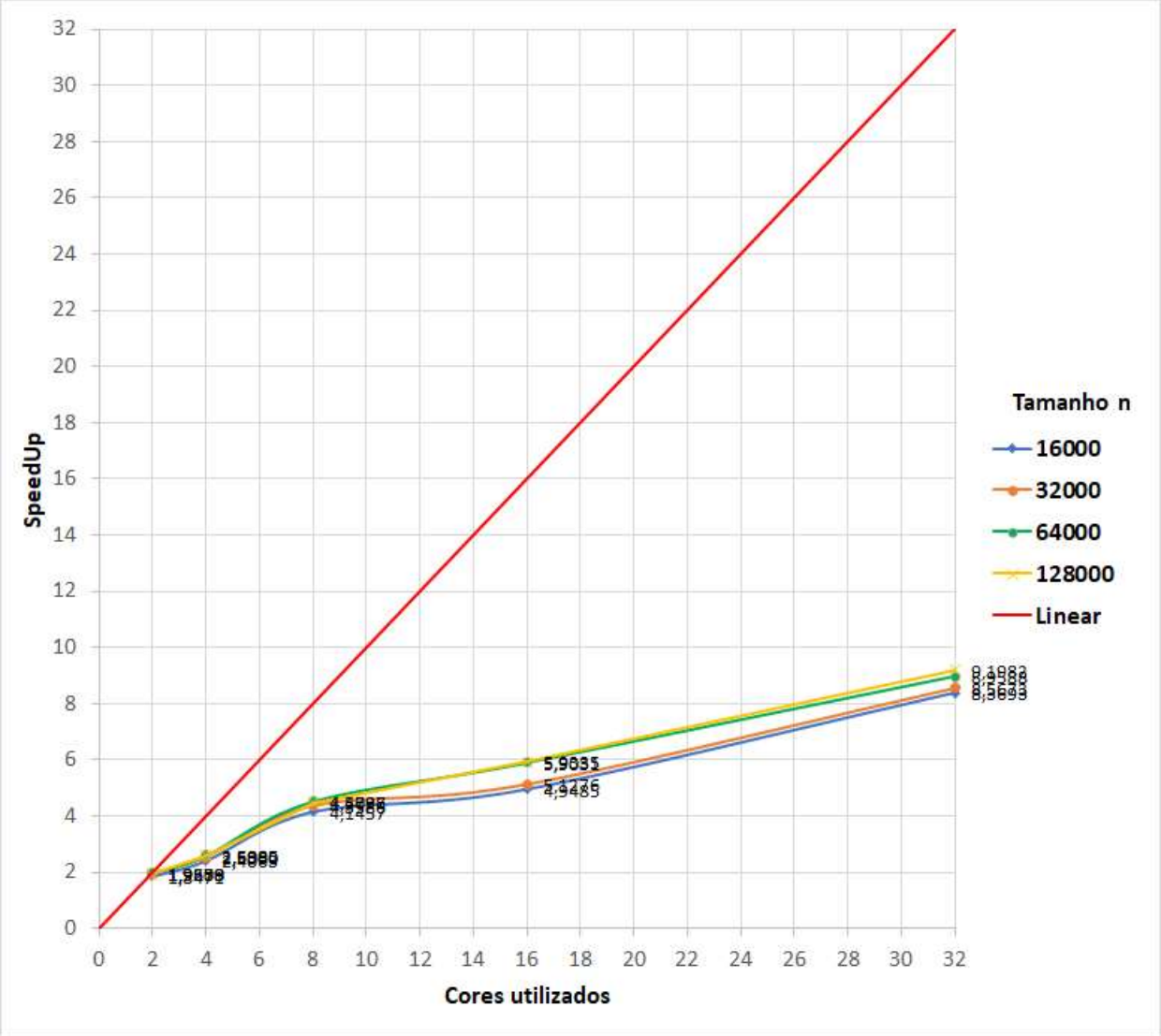
2.2.2 Análise de Escalabilidade (Speedup e Eficiência)

Após concluída a execução dos scripts, os arquivos txt de saída dos dados contendo o tempo das execuções e seus parâmetros foram importados para a o Excel, onde foi calculado a média de cada conjunto de 15 tempos, os valores de SpeedUp, e os valores de Eficiência.

N Cores	Tamanho	Média Tempo	SpeedUp	Eficiência
Serial	16000	0,801	--	--
Serial	32000	3,305	--	--
Serial	64000	13,635	--	--
Serial	128000	54,741	--	--
2	16000	0,433	1,8471	0,9236
2	32000	1,714	1,9279	0,9640
2	64000	6,943	1,9638	0,9819
2	128000	28,072	1,9500	0,9750
4	16000	0,333	2,4063	0,6016
4	32000	1,303	2,5360	0,6340
4	64000	5,243	2,6005	0,6501
4	128000	21,066	2,5986	0,6496
8	16000	0,193	4,1457	0,5182
8	32000	0,758	4,3586	0,5448
8	64000	3,023	4,5096	0,5637
8	128000	12,239	4,4727	0,5591
16	16000	0,162	4,9485	0,3093
16	32000	0,645	5,1276	0,3205
16	64000	2,310	5,9031	0,3689
16	128000	9,226	5,9335	0,3708
32	16000	0,096	8,3693	0,2615
32	32000	0,386	8,5673	0,2677
32	64000	1,522	8,9588	0,2800
32	128000	5,951	9,1982	0,2874

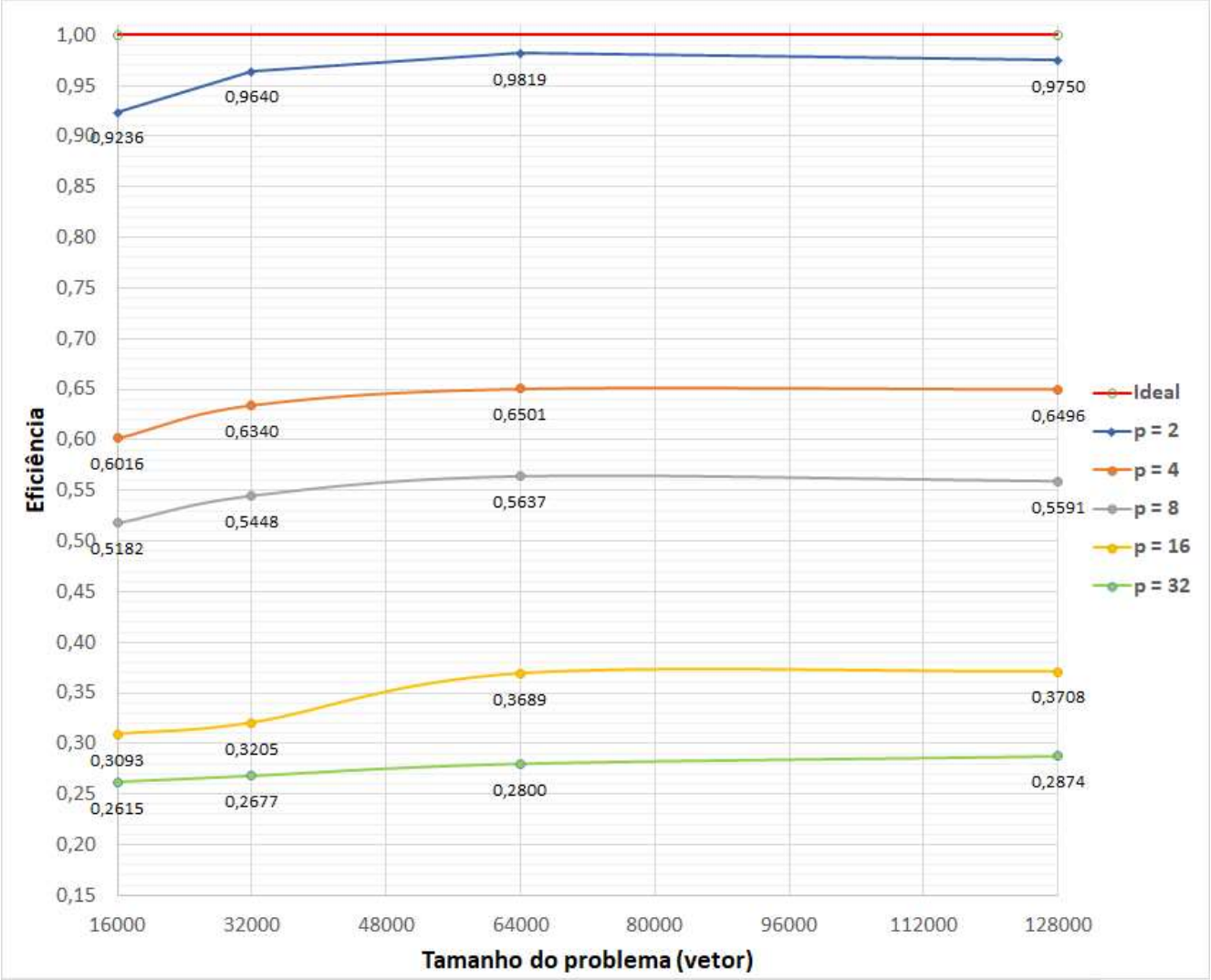
(tabela 1)

Os valores foram dispostos em dois gráficos, o primeiro contém as SpeedUps para os 4 tamanhos de problemas analisados, e mostra a evolução das SpeedUps em relação ao número de Cores utilizados:



(gráfico 1)

O segundo gráfico é o de Eficiência, analisado para os 5 valores de Cores utilizados na execução, e mostra a evolução da Eficiência em relação aos tamanhos do problema:



(gráfico 2)

Foi notado que quanto maior o número de cores utilizados, maior será o speedup e menor será a eficiência, e nos resultados experimentais esse padrão não teve nenhuma divergência.

O algoritmo é escalável quando executado com 16 e 32 cores, já para os tamanhos de 2, 4 e 8 cores, a eficiência reduz minimamente quando utilizamos o maior tamanho de vetor. Isso ocorre porque na prática, ao dobrar o tamanho do vetor, aumentamos em proporção logarítmica as interações de comparação dos números. Considerando isso, ainda assim podemos dizer que o algoritmo é escalável pois houve um grande ganho de tempo de execução do algoritmo paralelo em relação ao serial.

O algoritmo não é nem fortemente nem fracamente escalável seguindo os parâmetros de análise, e isso fica evidenciado na redução visível da eficiência quando utilizamos maior núcleos de processamento.

4. Conclusão.

A Lei de Gustafson é evidenciada e pode ser visualizada na tabela dos tempos, onde sempre que aumentamos o tamanho da região paralela do código, isso resulta em menor tempo de execução. Ao aumentarmos o tamanho do problema, como citado antes, aumentamos em proporção maior as interações, e isso por consequência, resulta em uma troca muito maior de quantidade de dados, que são transmitidos por protocolo de rede, isso gera grande impacto na eficiência da execução. Além disso, essas interações ocorrem por meio das funções Send e Recv da biblioteca MPI, ambas de característica bloqueante, o que gera espera nos processos, influenciando ainda mais no tempo de execução.