

Dividir e Conquistar: MergeSort Concorrente

Computação Concorrente – ICP-117 – 2022/1

Esteves Emmanuel Melo Ferreira - DRE: 117209640

João Pedro Souza - DRE: 119152051

Pedro Jorge Oliveira Câmara - DRE: 120182069

Dado um vetor v de números, ordenar v em ordem não decrescente.

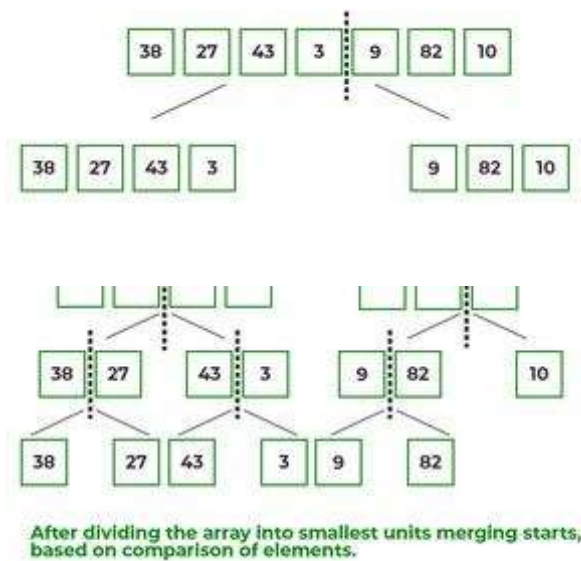
Solução proposta: MergeSort concorrente.



```
int[] Merge_sort(arr[], start, end)
    if(end>start):
        mid=(start+end)/2
        first[]=Merge_sort(arr, start, mid)
        second[]=Merge_sort(arr, mid+1, end)
        arr[]=Merge( first, second)
    return arr[]
```

Sabemos que há diversas soluções algorítmicas para o problema acima, sendo uma das mais famosas o algoritmo de ordenação MergeSort. Em geral, esses algoritmos clássicos possuem complexidade de tempo $O(n \log n)$ – o que também será verdade para a ordenação feita de forma concorrente. No entanto, o tempo será reduzido, devido à divisão de tarefas entre as threads.

De forma sequencial, o algoritmo MergeSort consiste em: dado um vetor v de números reais, recursivamente dividi-lo em dois, chamando novamente o MergeSort em cada metade, até que haja nenhum ou apenas um elemento, chegando-se ao caso base da recursão:



Após essa divisão, utilizamos a operação *merge*, que consiste na combinação de dois vetores menores, já ordenados, para formar um único vetor maior ordenado.

Dessa forma, o algoritmo sequencial em pseudocódigo é:

Entrada: vetor v , int $left$, int $right$

Saída: vetor v ordenado em ordem não descendente

if $left \geq right$:

return

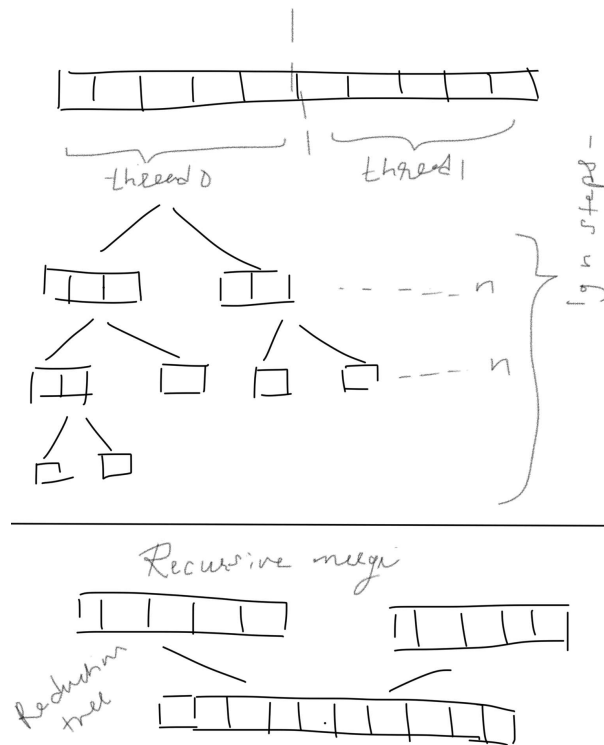
$mid \leftarrow (left + right) / 2$

$mergeSort(v, left, mid)$

$mergeSort(v, mid + 1, right)$

$merge(v, left, mid, right)$

Como o MergeSort é recursivo, parece promissor primeiro dividir o vetor v de entrada em aproximadamente n/t subvetores, para que cada thread possa executar o algoritmo independentemente em cada porção. No final, o algoritmo deve executar a operação de *merge* em v , obtendo assim o vetor ordenado.



Para verificar a corretude, há duas formas triviais:

- Criar dois vetores v_1 e v_2 , aplicar o MergeSort em v_1 , o MergeSort concorrente em v_2 e verificar se $v_1 = v_2$. No entanto, os vetores podem estar erradamente ordenados, mas serem iguais.

- Utilizar uma função auxiliar $is_sorted(v)$, que percorre v e verifica se $\forall e_i \in v, e_i \leq e_{i+1}$ para $i = [0, n - 1)$.

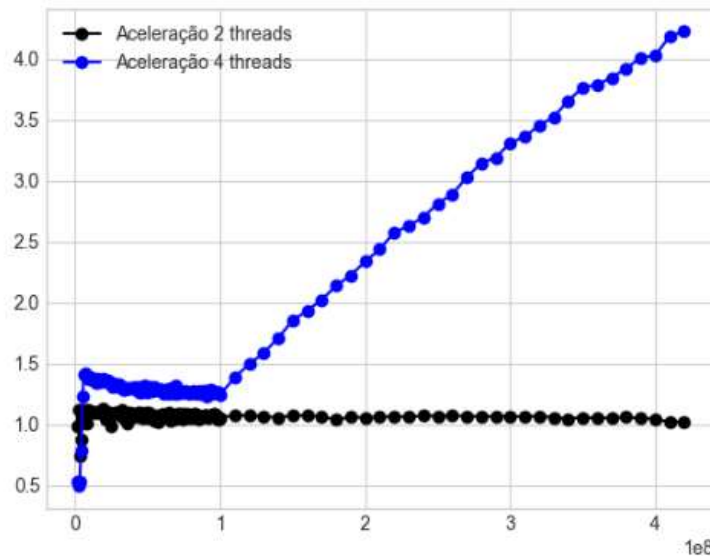
A segunda opção foi a escolhida, pois pode verificar identificar erros em ambas as ordenações independentemente, tanto sequencial como concorrente.

Os testes do algoritmo concorrente utilizaram 2 e 4 *threads*, usando um intervalo discreto entre 10^6 e $4 \cdot 10^8$.

A diferença de tempo começa a ficar expressiva quando $n > 200 \cdot 10^8$, como podemos observar no gráfico de tempo. Conforme n cresce após esse limiar, torna-se mais vantajoso o uso do *MergeSort* concorrente utilizando 4 *threads*.

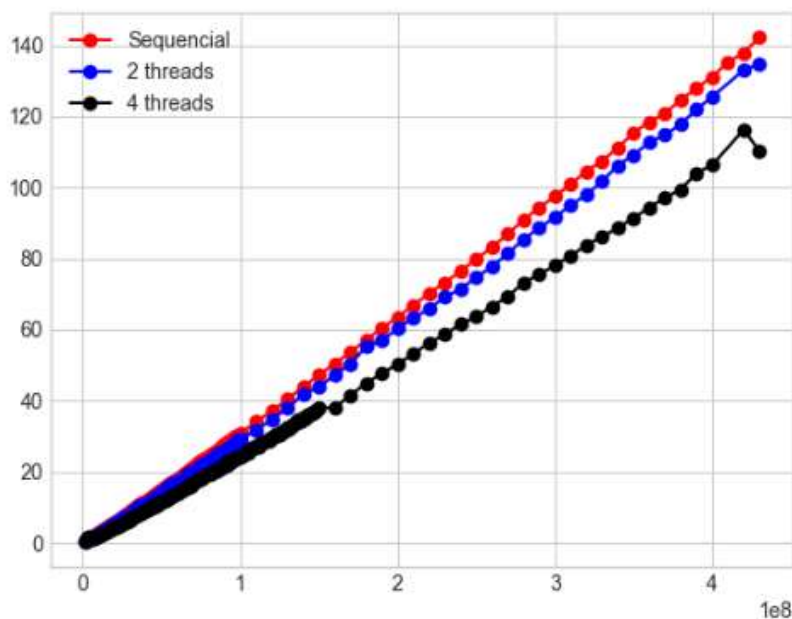
Apesar da complexidade de tempo não mudar, ou seja, permanecer $O(n \log n)$, é claro o ganho de desempenho obtido. Como estamos interessados somente na ordenação, descartamos a possibilidade de considerar a inicialização dos vetores, marcando apenas o tempo do algoritmo em si, pela fórmula $T_{sequencial} / t_p$:

(eixo x em centenas de milhões)



Pelo gráfico, percebemos que não há uma vantagem expressiva na utilização de 2 threads, enquanto que a aceleração obtida com 4 é ascendente a partir de $n = 10^8$.

Podemos também observar o gráfico que ilustra o tempo gasto em relação ao tamanho n da entrada (eixo x em centenas de milhões, eixo y em segundos):



Conforme n cresce para 10^9 , o ganho obtido vai tornando-se considerável. No entanto, para entradas menores que aproximadamente 10^8 , não haveria uma perda significativa de tempo em problemas que não requeressem uma computação de alto desempenho.

Os principais problemas enfrentados durante a execução dos testes foram:

- Aparentemente, o sistema operacional utilizado (*Ubuntu 20.08*) bloqueia o acesso excessivo à memória, o que fez com que obtivéssemos erros para a entrada $500 \cdot 10^8$. Para consertar, tivemos que modificar a *swap* da máquina, para que pudéssemos acessar mais

memória.

– Mesmo com acesso à memória, a máquina tratava por completo para entradas maiores que $400 \cdot 10^8$, o que tornou inviável testes para n maiores.

A ideia inicial foi implementar o algoritmo *QuickSort* de forma concorrente, mas sua natureza de difícil paralelização e falta de boas fontes de referência fez com que tomássemos o *MergeSort* como objetivo.

Máquina usada:

Intel i7 5500U: 2 cores, 4 threads

RAM: 8GB

Referências:

<https://www.geeksforgeeks.org/merge-sort/>

<https://malithjayaweera.com/2019/02/parallel-merge-sort/>

<https://www.geeksforgeeks.org/concurrent-merge-sort-in-shared-memory/>

<https://ark.intel.com/content/www/us/en/ark/products/85214/intel-core-i75500u-processor-4m-cache-up-to-3-00-ghz.html>