Universidade Federal do Rio Grande do Norte

Instituto Metrópole Digital

Análise e Comparação de Speedup e Eficiência do código serial e paralelo na ordenação de números utilizando MergeSort

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 Merge Sort***. As versões serial e paralela do algoritmo implementado serão explanadas, em seguida os resultados obtidos serão mostrados e comentados. Será também discutida 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.

Ordenação de números utilizando MergeSort

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 **Merge Sort** é um algoritmo de ordenação que se baseia no lema *Dividir e Conquistar*, isto é divide-se o problema em problemas menores, e ao resolver (conquistar), consegue realizar a tarefa desejada inicialmente. No caso, o **Merge Sort** divide o vetor a ser ordenado em 2 partes, as ordena chamando o próprio **Merge Sort** e, obtendo as duas partes ordenadas, realiza então o *merge* delas, isto é, mistura seus elementos de forma que a ordem desejada seja obtida. Por possuir essa natureza recursiva, o **Merge Sort** tende a possuir um alto gasto de memória, pois durante a operação de *merge*, é utilizada memória adicional a do vetor inicial. Assim, o algoritmo não é recomendado quando a economia de memória seja desejável. Porém o algoritmo possui uma complexidade de melhor, médio e pior caso igual: $O(n \cdot log_2 n)$. Existem diversas versões do **Merge Sort**, algumas são iterativas, não utilizando recursividade, outras não utilizam memória adicional, ou utilizam uma memória adicional menor do que a versão clássica. No quesito paralelização, diversas abordagens foram propostas ao longo dos anos, para essa atividade foi utilizada a ideia do *Dividir e Conquistar* de forma diferente do algoritmo serial, pois em vez de dividir o array em duas partes, é dividido em *n* partes, para então realizar o *merge* de cada uma delas em um grande vetor corretamente ordenado. Abordagens mais eficientes já foram propostas, porém de complexidade maior, por exemplo o *parallel merge*.

Desenvolvimento

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

Abordagens implementadas

Algoritmo Serial do MergeSort

O algoritmo inicia com a leitura dos seguintes argumentos:

- A **seed** a ser utilizada para preencher o vetor com números aleatórios;
- A **quantidade de números** a serem armazenados no vetor;
- A **flag** que sinaliza se o vetor ordenado deve ser exibido ou não;

Após ativar a **seed**, é alocado o devido espaço de memória para o vetor de números, em seguida preenche-se o vetor com os números pseudo-aleatórios utilizando a função **rand**.
Com o vetor pronto para ser ordenado, dá-se início ao **mergesort**, passando como argumentos o vetor a ser ordenado, a posição onde a ordenação deve iniciar e o tamanho do vetor. Essa função é apenas uma espécie de interface que encapsula a função interna do **mergesort**, onde os argumentos são: vetor, posição inicial e posição final. Assim, após subtrair uma unidade do tamanho do vetor, chama-se a função interna. A função interna tem início com uma condição que verifica se a posição inicial informada é menor que a posição final, pois se não for, não há nada a ordenar, assim, se os parâmetros atenderem essa condição, são realizados os seguintes passos:

1. Calcula-se a posição do elemento do meio do vetor, com base na posição inicial e final informadas;
2. Ordena o vetor da posição inicial até a posição do elemento do meio;
3. Ordena o vetor da posição do elemento do meio até a posição final;
4. Realiza o *merge* das metades ordenadas;

Como os passos 2 e 3 são as chamadas recursivas da função, não serão dados muitos detalhes, mas o passo 4, onde o *merge* ocorre, será explicado a seguir:
A função de *merge* recebe os seguintes parâmetros:

- O vetor onde o *merge* deve ocorrer;
- A posição inicial do vetor (inicio do primeiro vetor);
- A posição do meio do vetor (que delimita o final do primeiro vetor e início do segundo vetor);
- A posição final do vetor (final do segundo vetor);

Com esses parâmetros de posições são calculados os tamanhos de cada metade e com os devidos tamanhos, aloca-se a memória auxiliar do processo de *merge*.
Após copiar os valores do vetor original para os vetores auxiliares e inicializar as variáveis de apoio, que são:

- *i*: índice para as posições no primeiro vetor, inicia com 0;
- *j*: índice para as posições no segundo vetor, inicia com 0;
- *k*: índice para as posições no vetor ordenado, inicia com o valor da posição inicial informada por parâmetro;

É então iniciado um laço que continua enquanto as seguintes condições forem atendidas:
Se o valor de *i* for menor que o tamanho calculado para o primeiro vetor e o valor de *j* for menor que o tamanho calculado para o segundo vetor, isto é, enquanto existir elementos a percorrer em alguns dos dois vetores, o laço é executado.
Dentro do laço, é feita a seguinte verificação: O elemento da posição *i* do primeiro vetor **é menor** que o elemento da posição *j* do segundo vetor? Se ele for menor ele será colocado na posição *k* do vetor ordenado, mas se o menor elemento for o da posição *j*, então ele que é colocado na posição *k*. Após realizar essas operações são incrementados os índices envolvidos, isto é, o índice do elemento posicionado na posição *k* (*i* ou *j*) e o próprio *k*.
Após o término do laço, caso o valor de *i* seja menor que o tamanho do primeiro vetor, isto é, algum elemento do primeiro vetor não tenha sido posicionado no vetor ordenado, os elementos a partir da posição *i* até o final do primeiro vetor são copiados para o vetor ordenado, iniciando na posição *k*. Caso contrário, os elementos do segundo vetor que ainda não foram posicionados no vetor ordenado, que são copiados. Ao fim, ocorre a liberação da memória utilizada para armazenar os vetores auxiliares.

Algoritmo Paralelo do MergeSort

Na versão paralela, um novo parâmetro é necessário, a quantidade de threads a serem usadas, fora a leitura desse parâmetro, o inicio do algoritmo é o mesmo, aloca-se a memória do vetor e o preenche com os os números pseudo-aleatórios. A paralelização ocorre na função do **mergesort**, onde em vez de ser apenas uma interface para chamar o **mergesort** interno, na versão paralela ela é bem mais importante.
A função inicia calculando qual a quantidade de números a serem ordenados por cada thread, então aloca um vetor para armazenar os limites que definem os números a serem ordenados por cada thread, em seguida preenche esse vetor com base na posição inicial informada, o índice da posição e a quantidade de números que cada thread deve ordenar. Por fim, salienta-se que apenas o último limite não é calculado assim, pois ele é o próprio tamanho do vetor.
Com os limites devidamente calculados, define-se uma área paralela usando o ***#pragma omp parallel***, dentro dessa área é realizado um **omp for** onde cada thread executa a ordenação do vetor definido pelos limites calculados anteriormente, utilizando a versão serial do **mergesort**, assim, ao término desta etapa, temos diversos vetores ordenados,

porém o vetor completo ainda não está ordenado, dessa forma, é executada a função para fazer o *merge* de todos esses vetores em um único grande vetor.

Essa função recebe como parâmetros o vetor a ordenar, os limites, e a quantidade de vetores delimitados pelo vetor de limites. Caso sejam apenas 2 vetores, é realizada a mesma operação de *merge* utilizada pelo **mergesort** serial, caso sejam 3 vetores, são realizados dois *merges*, um entre o primeiro e o segundo vetor, e em seguida outro entre o resultado do primeiro *merge* com o terceiro vetor.

Caso a quantidade de vetores a passarem pelo *merge* seja maior que 3, então é iniciada uma região paralela com **omp sections**, duas **sections** são definidas, na primeira é feita a chamada recursiva da função de merge dos vetores, passando como parâmetros o próprio vetor a ordenar, os limites e a metade da quantidade de vetores a realizar o *merge*. Na segunda **section** outra chamada recursiva ocorre, novamente informando o vetor a ser ordenado, porém os limites agora informados iniciam a partir da outra metade de vetores, e a quantidade de vetores é novamente a metade de vetores da chamada atual da função.

Após o término dessas **sections** paralelas é verificado se a quantidade de vetores era par ou ímpar. Quando for uma quantidade par, é executado o *merge* dos vetores definidos pelo primeiro limite, o do meio e o final do vetor de limites, pois os vetores compreendidos nesses intervalos já tiverem o *merge* realizado nas chamadas recursiva da função de *merge*.

Caso a quantidade de vetores seja ímpar, é realizado o *merge* dos vetores definidos pelo limite inicial, o do meio e o penúltimo limite, após esse *merge* é executado outro *merge*, entre os vetores definidos pelo limite inicial, o penúltimo limite e o limite final. Após todos os *merges* ocorrerem o vetor está completamente ordenado.

Resultados encontrados

Corretude do Algoritmo Serial do MergeSort

A corretude da versão serial pode ser exemplificada com a seguinte instância do problema:

4, 1, 3, 2

Como o **mergesort** utiliza a estratégia de *Dividir* e *Conquistar*, o vetor é dividido em outros dois para então ser ordenado:

4, 1	3, 2

Após realizar essa divisão, como a condição ainda é satisfeita, isto é, o índice de início ainda é menor que o índice do fim, o vetor é dividido novamente.

4	1	3, 2

E com isso a chamada recursiva para de ocorrer, e o *merge* dos subvetores têm início. Assim após a execução do *merge*, obtém-se o seguinte resultado:

1, 4	3, 2

Após ter ordenado a primeira parte do vetor, é feito o ordenamento da segunda parte. Como a condição dos índices ainda é satisfeita, o vetor é dividido novamente:

1, 4	3	2

Neste momento a condição não é mais satisfeita, logo a chamada recursiva para de executar e a função de *merge* é realizada, tendo como resultado:

1, 4	2, 3

Com os dois subvetores devidamente ordenados, a chamada recursiva termina e então é executada a função de *merge* nos subvetores ordenados, resultando em:

1, 2, 3, 4

E assim, ocorre o ordenamento na versão serial do **mergesort**.

Corretude do Algoritmo Paralelo do MergeSort

A corretude da versão paralela pode ser exemplificada com a mesma instância do problema usada anteriormente, porém utilizando 2 threads:

4, 1, 3, 2

Como a versão paralela implementada do **mergesort** utiliza a estratégia de *Dividir* e *Conquistar* a nível de thread, o vetor é dividido em um número igual de subvetores para cada thread utilizada, no caso, é dividido em 2 subvetores, onde cada thread irá os ordenar seguindo a mesma estratégia utilizada na versão serial.

4, 1	3, 2

Após cada thread ter ordenado seu subvetor, temos o seguinte resultado:

1, 4	2, 3

Nesse momento é feito o *merge* dos vetores ordenados por cada thread num único vetor final.

Como são duas threads, temos apenas duas partes para fazer o merge, após executar a função, obtemos o seguinte resultado:

1, 2, 3, 4

E assim, ocorre o ordenamento na versão paralela do **mergesort**. Obs.: De acordo com o número de threads, serão necessárias mais operações de *merge* no final.

Análise de Speedup, Eficiência e Escalabilidade

Abaixo está a tabela 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.

Tamanho do Problema	Serial	4	8	16	32
100000000	36,29818182	11,66203213	7,16241470	5,83763487	5,27261145
150000000	55,96000000	17,79348821	10,91436425	8,56394576	7,81364857
200000000	74,89909091	23,89023680	14,62228583	14,16228459	9,95496954
250000000	94,95818182	29,98619589	18,39116566	13,42333262	12,98262784
300000000	115,28272727	36,43567126	22,27351928	18,93479857	16,57769051
350000000	135,58272727	42,74521834	26,03315648	21,28726016	18,59280132

Tabela 1: Apresenta os tempos médios obtidos nos experimentos por tamanho do problema e algoritmo utilizado, assim como a quantidade de cores na versão paralela.

Speedup 4	Speedup 8	Speedup 16	Speedup 32	Eficiência 4	Eficiência 8	Eficiência 16	Eficiência 32
3,11250916	5,06786933	6,21796029	6,88428916	0,77812729	0,63348367	0,38862252	0,21513404
3,14497075	5,12718824	6,53437113	7,16182709	0,78624269	0,64089853	0,40839820	0,22380710
3,13513388	5,12225597	5,28863055	7,52378906	0,78378347	0,64028200	0,33053941	0,23511841
3,16672986	5,16324977	7,07411375	7,31424970	0,79168246	0,64540622	0,44213211	0,22857030
3,16400723	5,17577514	6,08840526	6,95408852	0,79100181	0,64697189	0,38052533	0,21731527
3,17188056	5,20807868	6,36919577	7,29221621	0,79297014	0,65100984	0,39807474	0,22788176

Tabela 2: Apresenta os valores do speedup e da eficiência de 4 a 32 cores, seguindo a mesma ordem dos tamanhos de problema da **Tabela 1**.

Com a tabela dos tempos, pode-se perceber que a versão paralela consegue reduzir consideravelmente o tempo gasto para realizar a ordenação, porém ao observar a **Tabela 2**, percebe-se que o speedup diminui o seu aumento conforme o aumento do número de cores. A eficiência reduz rapidamente conforme o aumento de cores. Esse comportamento pode ser melhor entendido nos gráficos abaixo, onde é feita a comparação de speedup e eficiência do **mergesort** implementado.

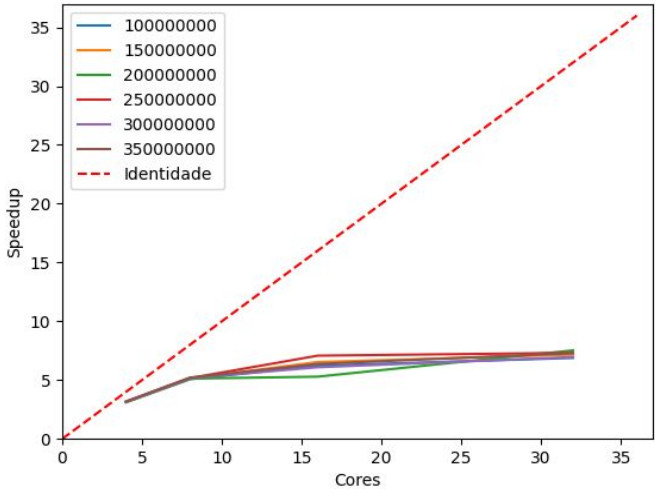
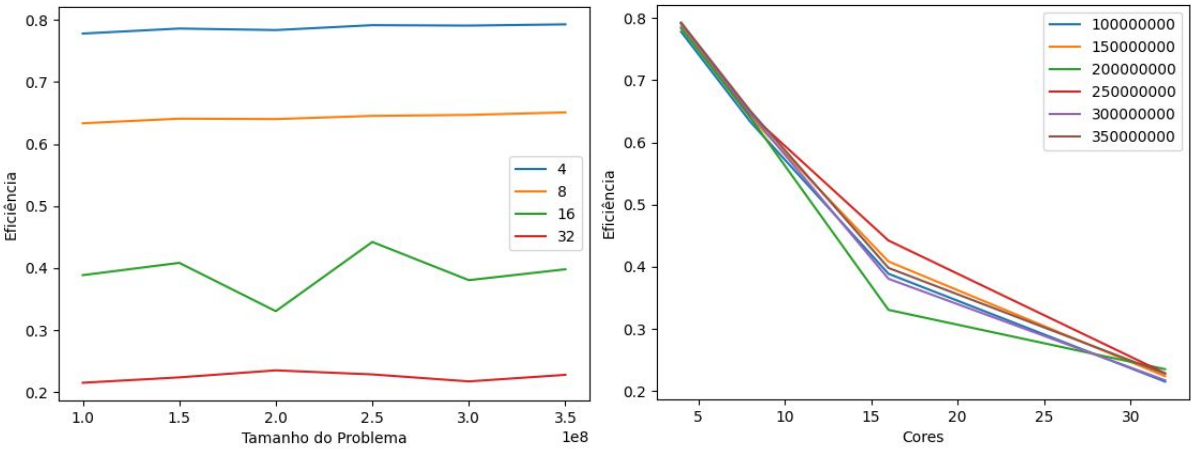


Imagem 1: Comparações do speedup em cada um dos tamanhos do problema com o speedup ideal.



Imagens 2 e 3: Apresenta a eficiência por cores em cada tamanho de problema utilizado e compara a eficiência por tamanho do problema, de acordo com a quantidade de cores.

Observando o gráfico da **imagem 1**, relativo ao speedup, pode-se observar que em um dado momento é como se o speedup tivesse alcançado um platô, não conseguindo aumentar mais. Provavelmente esse é o limite que essa abordagem de paralelização pode alcançar, talvez utilizando outras técnicas sejam obtidos resultados melhores. Em relação aos gráficos relativos à eficiência, pode-se observar um comportamento bem comum, a eficiência se mantém constante conforme aumenta-se o tamanho do problema para uma mesma quantidade de cores, e conforme o número de cores aumenta, para um mesmo tamanho de problema, a eficiência diminui.

Assim, analisando esses gráficos, a versão do **mergesort** implementada pode ser classificada como **Fracamente Escalável**, visto que com o aumento do tamanho do problema e do número de cores em proporções iguais, a eficiência se mantém constante.

Considerações Finais

Neste relatório apresentou-se brevemente o problema da ordenação de um vetor de números e o que é o algoritmo de ordenação MergeSort . Foram explanadas as versões serial e paralela do algoritmo implementado, explicando como cada versão utiliza a ideia de *Dividir e Conquistar*.

Foi realizada uma breve apresentação da corretude dos algoritmos implementados, exemplificando a execução dos mesmos por meio de uma pequena instância do problema, nessa mesma instância foi brevemente simulada a execução do algoritmo serial, e em seguida o algoritmo paralelo com apenas duas threads. Feito isso, foi exposto a análise e comparação do speedup e eficiências do algoritmo paralelo implementado, apresentando os tempos médios obtidos nos experimentos realizados e discutindo os gráficos de speedup e eficiência gerados por esses tempos. Por fim, foi feita a categorização da versão paralela em relação a escalabilidade, com Fracamente Escalável sendo o escolhido, dados o comportamento observado no gráfico da **Figura 2**.

A apresentação do trabalho pode ser visualizada nest link: https://youtu.be/7fqCDI79TME

Referências

Introdução a Sistemas Paralelos

https://en.wikipedia.org/wiki/Merge_sort

https://en.wikipedia.org/wiki/Merge_algorithm#Parallel_merge

https://pt.wikipedia.org/wiki/Merge_sort

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